## Python Pandas Introduction

Data analysis requires lots of processing, such as **restructuring, cleaning** or **merging**, etc. There are different tools are available for fast data processing, such as **Numpy, Scipy, Cython**, and **Panda**. But we prefer Pandas because working with Pandas is fast, simple and more expressive than other tools.

Pandas is built on top of the **Numpy** package, means **Numpy** is required for operating the Pandas.

## Key Features of Pandas

* It has a fast & efficient DataFrame object with default and customized indexing.
* Used for reshaping and pivoting of the data sets.
* Group by data for aggregations and transformations.
* It is used for data alignment and integration of the missing data.
* Provide the functionality of Time Series.
* Process a variety of data sets in different formats like **matrix data, tabular heterogeneous, time series.**
* Handle multiple operations of the data sets such as **subsetting**, **slicing**, **filtering**, groupBy, re-ordering, and re-shaping.
* It integrates with the other libraries such as **SciPy**, and **scikit-learn.**
* Provides fast performance, and If you want to speed it, you can use **Cython**.

## Benefits of Pandas

* **Data Representation:** It represents the data in a form that is suited for data analysis through its DataFrame and Series.
* **Clear code:** The clear API of the Pandas allows you to focus on the core part of the code. So, it provides clear and concise code for the user.

## Python Pandas Data Structure

The Pandas provides two data structures for processing the data, i.e., **Series** and **DataFrame**, which are discussed below:

### 1) Series

It is defined as a one-dimensional array that is capable of storing various data types. The row labels of series are called the **index**. We can easily convert the list, tuple, and dictionary into series using "series' method. A Series cannot contain multiple columns. It has one parameter:

**Data:** It can be any list, dictionary, or scalar value.

**Creating Series from Array:**

Before creating a Series, Firstly, we have to import the numpy module and then use array() function in the program.

1. **import** pandas as pd
2. **import** numpy as np
3. info = np.array(['P','a','n','d','a','s'])
4. a = pd.Series(info)
5. **print**(a)

## Python Pandas DataFrame

It is a widely used data structure of pandas and works with a two-dimensional array with labeled axes (rows and columns). DataFrame is defined as a standard way to store data and has two different indexes, i.e., row index and column index.

* The columns can be heterogeneous types like int, bool, and so on.
* It can be seen as a dictionary of Series structure where both rows & columns are indexed. It is denoted as "columns" in case of columns & "index" in case of rows.

**Create a DataFrame using List:**

1. **import pandas as pd**
2. **x = ['Python', 'Pandas'] # a list of strings**
3. **df = pd.DataFrame(x) # Calling DataFrame constructor on list**
4. **print(df)**

# Python Pandas Series

The Pandas Series can be defined as a one-dimensional array that is capable of storing various data types. We can easily convert the list, tuple, and dictionary into a series using "**series**' method. The row labels of series are called the index. A Series cannot contain multiple columns. It has the following parameter:

* **data:** It can be any list, dictionary, or scalar value.
* **index:** The value of index should be unique and hashable. It must be of the same length as data. If we do not pass any index, default **np.arrange(n)** will be used.
* **dtype:** It refers to the data type of series.
* **copy:** It is used for copying the data.

## Creating a Series: We can create a Series in two ways:

1. Create an empty Series
2. Create a Series using inputs.

**1)**We can easily create an empty series in Pandas,means it will not have any value.

1. **import** pandas as pd
2. x = pd.Series()
3. print (x)

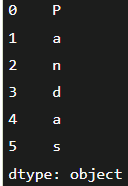
**2)**We can create Series by using various inputs: **Array,Dict,Scalar Value.**

**Creating Series from Array:**

Before creating a Series, firstly, we have to import the **numpy** module and then use array() function in the program. If the data is ndarray, then the passed index must be of the same length.

If we do not pass an index, then by default an index of **range(n)** is being passed where n defines the length of an array, i.e., [0,1,2,....**range(len(array))-1**].

1. **import** pandas as pd
2. **import** numpy as np
3. info = np.array(['P','a','n','d','a','s'])
4. a = pd.Series(info)
5. print(a)

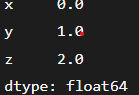


**Create a Series from dict**

We can also create a Series from dict. **If the dictionary object is being passed as an input and the index is not specified, then the dictionary keys are taken in a sorted order to construct the index**.

If index is passed, then values correspond to a particular label in the index will be extracted from the **dictionary**.

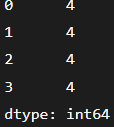
1. **import** pandas as pd
2. **import** numpy as np
3. info = {'x' : 0., 'y' : 1., 'z' : 2.}
4. a = pd.Series(info)
5. print (a)



**Create a Series using Scalar:**

If we take the scalar values, then the index must be provided. The scalar value will be repeated for matching the length of the index.

1. **import** pandas as pd
2. **import** numpy as np
3. x = pd.Series(4, index=[0, 1, 2, 3])
4. print (x)



## Accessing data from series with Position:

Once you create the Series type object, you can access its indexes, data, and even individual elements.

The data in the Series can be accessed similar to that in the ndarray.

1. **import** pandas as pd
2. x = pd.Series([1,2,3],index = ['a','b','c'])
3. print (x[0]) //1 #retrieve the first element

### Series object attributes

The Series attribute is defined as any information related to the Series object such as size, datatype. etc. Below are some of the attributes .

| **Attributes** | **Description** |
| --- | --- |
| **Series.index** | Defines the index of the Series. |
| **Series.shape** | It returns a tuple of shape of the data. |
| **Series.dtype** | It returns the data type of the data. |
| **Series.size** | It returns the size of the data. |
| **Series.empty** | It returns True if Series object is empty, otherwise returns false. |
| **Series.hasnans** | It returns True if there are any NaN values, otherwise returns false. |
| **Series.nbytes** | It returns the number of bytes in the data. |
| **Series.ndim** | It returns the number of dimensions in the data. |
| **Series.itemsize** | It returns the size of the datatype of item. |

### Retrieving Index array and data array of a series object

We can retrieve the index array and data array of an existing Series object by using the attributes index and values.

1. **import** numpy as np
2. **import** pandas as pd
3. x=pd.Series(data=[2,4,6,8])
4. y=pd.Series(data=[11.2,18.6,22.5], index=['a','b','c'])
5. print(x.index) //RangeIndex(start=0, stop=4, step=1)
6. print(x.values) //[2 4 6 8]
7. print(y.index) //Index(['a', 'b', 'c'], dtype='object')
8. print(y.values) //[11.2 18.6 22.5]

### Retrieving Types (dtype) and Size of Type (itemsize)

You can use attribute dtype with Series object as <objectname> dtype for retrieving the data type of an individual element of a series object, you can use the **itemsize** attribute to show the number of bytes allocated to each data item.

1. a=pd.Series(data=[1,2,3,4])
2. b=pd.Series(data=[4.9,8.2,5.6],
3. index=['x','y','z'])
4. print(a.dtype) //int64
5. print(a.itemsize) //8
6. print(b.dtype) //float64
7. print(b.itemsize) //8

### Retrieving Shape

The shape of the Series object defines total number of elements including missing or empty values(NaN).

1. a=pd.Series(data=[1,2,3,4])
2. b=pd.Series(data=[4.9,8.2,5.6],index=['x','y','z'])
3. print(a.shape) //(4,)
4. print(b.shape) //(3,)

### Retrieving Dimension, Size and Number of bytes:

1. a=pd.Series(data=[1,2,3,4])
2. b=pd.Series(data=[4.9,8.2,5.6],
3. index=['x','y','z'])
4. print(a.ndim, b.ndim) // 1 1
5. print(a.size, b.size) // 4 3
6. print(a.nbytes, b.nbytes) // 32 24

### Checking Emptiness and Presence of NaNs

To check the Series object is empty, you can use the **empty attribute**. Similarly, to check

if a series object contains some NaN values or not, you can use the **hasans** attribute.

1. a=pd.Series(data=[1,2,3,np.NaN])
2. b=pd.Series(data=[4.9,8.2,5.6],index=['x','y','z'])
3. c=pd.Series()
4. print(a.empty,b.empty,c.empty) //False False True
5. print(a.hasnans,b.hasnans,c.hasnans) //True False False
6. print(len(a),len(b)) // 4 3
7. print(a.count( ),b.count( )) // 3 3

## Series Functions

| **Functions** | **Description** |
| --- | --- |
| [Pandas Series.map()](https://www.javatpoint.com/pandas-series-map) | Map the values from two series that have a common column. |
| [Pandas Series.std()](https://www.javatpoint.com/pandas-standard-deviation) | Calculate the standard deviation of the given set of numbers, DataFrame, column, and rows. |
| [Pandas Series.to\_frame()](https://www.javatpoint.com/pandas-series-to_frame) | Convert the series object to the dataframe. |
| [Pandas Series.value\_counts()](https://www.javatpoint.com/pandas-series-value_counts) | Returns a Series that contain counts of unique values. |

# Pandas Series.map()

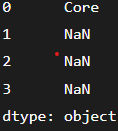
The main task of map() is used to map the values from two series that have a common column. To map the two Series, the last column of the first Series should be the same as the index column of the second series, and the values should be unique.

**Series.map(arg, na\_action=None)**

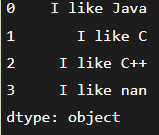
* **arg:** function, dict, or Series.It refers to the mapping correspondence.
* **na\_action:** {None, 'ignore'}, Default value None. If ignore, it returns null values, without passing it to the mapping correspondence.

It returns the Pandas Series with the same index as a caller.

1. a = pd.Series(['Java', 'C', 'C++', np.nan])
2. a.map({'Java': 'Core'})



1. a = pd.Series(['Java', 'C', 'C++', np.nan])
2. a.map({'Java': 'Core'})
3. a.map('I like {}'.format, na\_action='ignore')



# Pandas Series.std()

The Pandas **std()** is defined as a function for calculating the standard deviation of given set of numbers, DataFrame, column, and rows. In respect to calculate the standard deviation, we need to import the package named "**statistics**" for calculation of median.

The standard deviation is normalized by N-1 by default and can be changed using the **ddof** argument.

**Series.std(axis=None, skipna=None, level=None, ddof=1, numeric\_only=None, \*\*kwargs)**

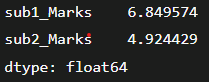
* **axis:** {index (0), columns (1)}
* **skipna:** It excludes all the NA/null values. If NA is present in an entire row/column, the result will be NA.
* **level:** It counts along with a particular level, and collapsing into a scalar if the axis is a MultiIndex (hierarchical).
* **ddof:** Delta Degrees of Freedom. The divisor used in calculations is N - ddof, where N represents the number of elements.
* **numeric\_only:** boolean, default value None  
  It includes only float, int, boolean columns. If it is None, it will attempt to use everything, so use only numeric data.  
  It is not implemented for a Series.

**It returns Series or DataFrame if the level is specified.**

1. **import** pandas as pd
2. **import** numpy as np
3. print(np.std([4,7,2,1,6,3])) //2.1147629234082532
4. print(np.std([6,9,15,2,-17,15,4])) //10.077252622027656

Ex:2 #Create a DataFrame

1. info = {
2. 'Name':['Parker','Smith','John','William'],
3. 'sub1\_Marks':[52,38,42,37],
4. 'sub2\_Marks':[41,35,29,36]}
5. data = pd.DataFrame(info)
6. data.std() # standard deviation of the dataframe



# Pandas Series.to\_frame()

Series is defined as a type of list that can hold an integer, string, double values, etc. It returns an object in the form of a list that has an index starting from 0 to n where n represents the length of values in Series.

The main difference between Series and Data Frame is that

**Series can only contain a single list with a particular index, whereas the DataFrame is a combination of more than one series that can analyze the data**.

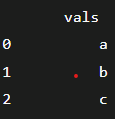
Pandas **Series.to\_frame()** function is used to convert the series object to DataFrame.

**Series.to\_frame(name=None)**

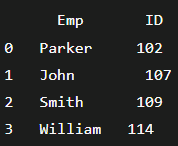
**name:** Refers to the object. Its Default value is None. If it has one value, the passed name will be substituted for the series name.

**It returns DataFrame representation of Series.**

1. s = pd.Series(["a", "b", "c"],
2. name="vals")
3. s.to\_frame()



1. emp = ['Parker', 'John', 'Smith', 'William']
2. id = [102, 107, 109, 114]
3. emp\_series = pd.Series(emp)
4. id\_series = pd.Series(id)
5. frame = { 'Emp': emp\_series, 'ID': id\_series }
6. result = pd.DataFrame(frame)
7. print(result)



# Pandas Series.unique()

While working with the DataFrame in Pandas, you need to find the **unique** elements present in the column. For this, we have to use unique() method to extract the unique values from columns.Pandas library in Python can easily help us to find unique data.

The unique values present in columns are returned in order of its occurrence. This does not sort the order of its appearance. In addition, this method is based on the **hash-table**.

It is significantly faster than **numpy.unique()** method and also includes null values.

**pandas.unique(values)**

This method returns a numpy.ndarray or ExtensionArray object and can be:

* **index:** Returns when user passes index as an input.
* **Categorical:** Returns when user passes a Categorical dtype as an input.
* **ndarray:** Returns when user passes a ndarray/Series as an input.

AD

1. **import** pandas as pd
2. pd.unique(pd.Series([2, 1, 3, 3])) //[2,1,3]

# Pandas Series.value\_counts()

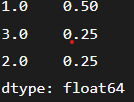
The value\_counts() function returns a Series that contain counts of unique values. It returns an object that will be in descending order so that its first element will be the most frequently-occurred element.

By default, it excludes NA values.

**Series.value\_counts(normalize=False, sort=True, ascending=False, bins=None, dropna=True)**

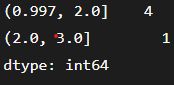
* **normalize:** If it is true, then the returned object will contain the relative frequencies of the unique values.
* **sort:** It sort by the values.
* **ascending:** It sort in the ascending order.
* **bins:** Rather than counting the values, it groups them into the half-open bins that provide convenience for the pd.cut, which only works with numeric data.
* **dropna:** It does not include counts of NaN.

1. index = pd.Index([2, 1, 1, np.nan, 3])
2. a = pd.Series([2, 1, 1, np.nan, 3])
3. a.value\_counts(normalize=True)

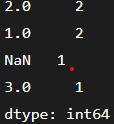


Ex: 1. a = pd.Series([1, 3, 2, 2, 1, np.nan])

2.a.value\_counts(bins=2)

It creates the group. Like as count b/w value 2 & 3 is 1.

1. a = pd.Series([1, 3, 2, 2, 1, np.nan])
2. a.value\_counts(dropna=False) //It also count NaN



# Python Pandas DataFrame

Pandas DataFrame is a widely used data structure which works with a two-dimensional array with labeled axes (rows and columns). DataFrame is defined as a standard way to store data that has two different indexes, i.e., **row index** and **column index**. It consists of the following properties:

* The columns can be heterogeneous types like int, bool, and so on.
* It can be seen as a dictionary of Series structure where both the rows and columns are indexed. It is denoted as "columns" in case of columns and "index" in case of rows.

## Parameter & Description:

**data:** It consists of different forms like ndarray, series, map, constants, lists, array.

**index:** The Default np.arrange(n) index is used for the row labels if no index is passed.

**columns:** The default syntax is np.arrange(n) for the column labels. It shows only true if no index is passed.

**dtype:** It refers to the data type of each column.

**copy():** It is used for copying the data.

## Create a DataFrame

We can create a DataFrame using **dict, Lists, Numpy ndarrrays, Series.**

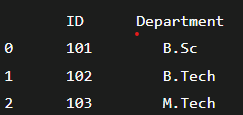
1. **import** pandas as pd
2. df = pd.DataFrame() //create an empty data frame
3. **print** (df)

Ex:

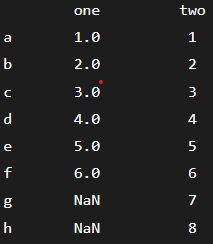
1. x = ['Python', 'Pandas']
2. df = pd.DataFrame(x) # Calling DataFrame constructor on list
3. **print**(df)

Ex:

1. info = {'ID' :[101, 102, 103],'Department' :['B.Sc','B.Tech','M.Tech',]}
2. df = pd.DataFrame(info)
3. **print** (df)

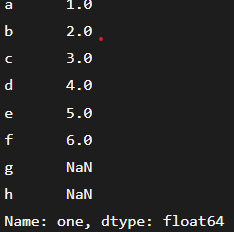


1. info = {'one' : pd.Series([1, 2, 3, 4, 5, 6], index=['a', 'b', 'c', 'd', 'e', 'f']),
2. 'two' : pd.Series([1, 2, 3, 4, 5, 6, 7, 8], index=['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'])}
3. d1 = pd.DataFrame(info)
4. **print** (d1)



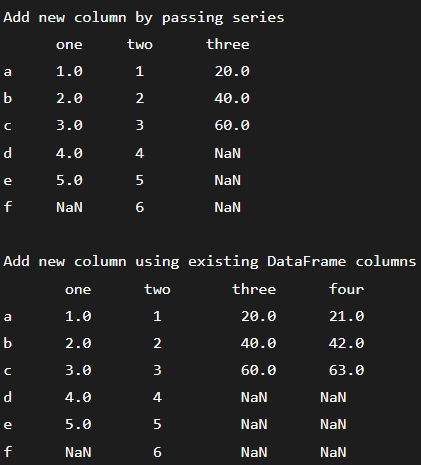
## Column Selection:We can select any column from the DataFrame.

1. info = {'one' : pd.Series([1, 2, 3, 4, 5, 6], index=['a', 'b', 'c', 'd', 'e', 'f']),
2. 'two' : pd.Series([1, 2, 3, 4, 5, 6, 7, 8], index=['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'])}
3. d1 = pd.DataFrame(info)
4. **print** (d1 ['one'])



## Column Addition:We can also add any new column to an existing DataFrame.

1. info = {'one' : pd.Series([1, 2, 3, 4, 5], index=['a', 'b', 'c', 'd', 'e']),
2. 'two' : pd.Series([1, 2, 3, 4, 5, 6], index=['a', 'b', 'c', 'd', 'e', 'f'])}
3. df = pd.DataFrame(info)
4. df['three']=pd.Series([20,40,60],index=['a','b','c']) #Add new colmn by passing seris
5. **print** (df)
6. df['four']=df['one']+df['three'] //Add new column using existing DataFrame colum
7. **print** (df)



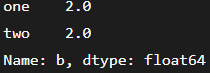
## Column Deletion:We can delete any column from the existing DataFrame.

1. **import** pandas as pd
2. info = {'one' : pd.Series([1, 2], index= ['a', 'b']),
3. 'two' : pd.Series([1, 2, 3], index=['a', 'b', 'c'])}
4. df = pd.DataFrame(info)
5. **print** (df)
6. **del** df['one'] //Delete the first column using del function
7. **print** (df)
8. df.pop('two') # Delete the another column using pop function
9. **print** (df)

### Row Selection: We can easily select, add, or delete any row at anytime. First of all, we will understand the row selection.

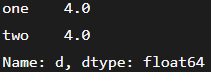
**Selection by Label:** We can select any row by passing the row label to a **loc** function.

1. info = {'one' : pd.Series([1, 2, 3, 4, 5], index=['a', 'b', 'c', 'd', 'e']),
2. 'two' : pd.Series([1, 2, 3, 4, 5, 6], index=['a', 'b', 'c', 'd', 'e', 'f'])}
3. df = pd.DataFrame(info)
4. **print** (df.loc['b'])



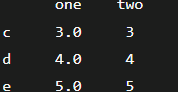
**Selection by integer location:** The rows can also be selected by passing the integer location to an **iloc** function.

1. info = {'one' : pd.Series([1, 2, 3, 4, 5], index=['a', 'b', 'c', 'd', 'e']),
2. 'two' : pd.Series([1, 2, 3, 4, 5, 6], index=['a', 'b', 'c', 'd', 'e', 'f'])}
3. df = pd.DataFrame(info)
4. **print** (df.iloc[3])



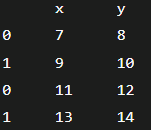
**Slice Rows:-** It is another method to select multiple rows using **':'** operator.

1. info = {'one' : pd.Series([1, 2, 3, 4, 5], index=['a', 'b', 'c', 'd', 'e']),
2. 'two' : pd.Series([1, 2, 3, 4, 5, 6], index=['a', 'b', 'c', 'd', 'e', 'f'])}
3. df = pd.DataFrame(info)
4. **print** (df[2:5])



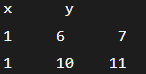
**Addition of rows:** We can easily add new rows to the DataFrame using **append** function. It adds the new rows at the end.

1. d = pd.DataFrame([[7, 8], [9, 10]], columns = ['x','y'])
2. d2 = pd.DataFrame([[11, 12], [13, 14]], columns = ['x','y'])
3. d = d.append(d2)
4. **print** (d)



**Deletion of rows:** We can delete or drop any rows from a DataFrame using the **index** label. If in case, the label is duplicate then multiple rows will be deleted.

1. a\_info = pd.DataFrame([[4, 5], [6, 7]], columns = ['x','y'])
2. b\_info = pd.DataFrame([[8, 9], [10, 11]], columns = ['x','y'])
3. a\_info = a\_info.append(b\_info)
4. a\_info = a\_info.drop(0) # Drop rows with label 0



## 

## DataFrame Functions

| **Functions** | **Description** |
| --- | --- |
| [Pandas DataFrame.append()](https://www.javatpoint.com/pandas-append) | Add the rows of other dataframe to the end of the given dataframe. |
| [Pandas DataFrame.apply()](https://www.javatpoint.com/pandas-apply) | Allows the user to pass a function and apply it to every single value of the Pandas series. |
| [Pandas DataFrame.assign()](https://www.javatpoint.com/pandas-dataframe-assign) | Add new column into a dataframe. |
| [Pandas DataFrame.astype()](https://www.javatpoint.com/pandas-dataframe-astype) | Cast the Pandas object to a specified dtype.astype() function. |
| [Pandas DataFrame.concat()](https://www.javatpoint.com/pandas-concatenation) | Perform concatenation operation along an axis in the DataFrame. |
| [Pandas DataFrame.count()](https://www.javatpoint.com/pandas-count) | Count the number of non-NA cells for each column or row. |
| [Pandas DataFrame.describe()](https://www.javatpoint.com/pandas-dataframe-describe) | Calculate some statistical data like percentile, mean and std of the numerical values of the Series or DataFrame. |
| [Pandas DataFrame.drop\_duplicates()](https://www.javatpoint.com/pandas-dataframe-drop_duplicates) | Remove duplicate values from the DataFrame. |
| [Pandas DataFrame.groupby()](https://www.javatpoint.com/pandas-groupby) | Split the data into various groups. |
| [Pandas DataFrame.head()](https://www.javatpoint.com/pandas-dataframe-head) | Returns the first n rows for the object based on position. |
| [Pandas DataFrame.hist()](https://www.javatpoint.com/pandas-dataframe-hist) | Divide the values within a numerical variable into "bins". |
| [Pandas DataFrame.iterrows()](https://www.javatpoint.com/pandas-dataframe-iterrows) | Iterate over the rows as (index, series) pairs. |
| [Pandas DataFrame.mean()](https://www.javatpoint.com/pandas-dataframe-mean) | Return the mean of the values for the requested axis. |
| [Pandas DataFrame.melt()](https://www.javatpoint.com/pandas-melt) | Unpivots the DataFrame from a wide format to a long format. |
| [Pandas DataFrame.merge()](https://www.javatpoint.com/pandas-merge) | Merge the two datasets together into one. |
| [Pandas DataFrame.pivot\_table()](https://www.javatpoint.com/pandas-pivot-table) | Aggregate data with calculations such as Sum,Count,Average,Max, Min. |
| [Pandas DataFrame.query()](https://www.javatpoint.com/pandas-dataframe-query) | Filter the dataframe. |
| [Pandas DataFrame.sample()](https://www.javatpoint.com/pandas-dataframe-sample) | Select the rows and columns from the dataframe randomly. |
| [Pandas DataFrame.shift()](https://www.javatpoint.com/pandas-shift) | Shift column or subtract the column value with the previous row value from the dataframe. |
| [Pandas DataFrame.sort()](https://www.javatpoint.com/python-pandas-sorting) | Sort the dataframe. |
| [Pandas DataFrame.sum()](https://www.javatpoint.com/pandas-sum) | Return the sum of the values for the requested axis by the user. |
| [Pandas DataFrame.to\_excel()](https://www.javatpoint.com/pandas-dataframe-to_excel) | Export the dataframe to the excel file. |
| [Pandas DataFrame.transpose()](https://www.javatpoint.com/pandas-dataframe-transpose) | Transpose the index and columns of the dataframe. |
| [Pandas DataFrame.where()](https://www.javatpoint.com/pandas-dataframe-where) | Check the dataframe for one or more conditions. |

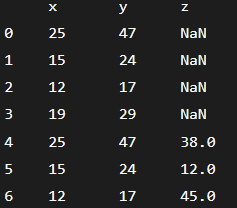
# Pandas DataFrame.append()

The Pandas **append()** function is used to add the rows of other dataframe to the end of the given dataframe, returning a new dataframe object. The new columns and the new cells are inserted into the original DataFrame that are populated with NaN value.

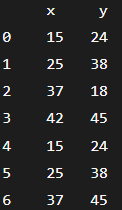
**DataFrame.append(other, ignore\_index=False, verify\_integrity=False, sort=None)**

* **other:** DataFrame or Series/dict-like object, or a list of these It refers to the data to be appended.
* **ignore\_index:** If it is true, it does not use the index labels.
* **verify\_integrity:** If it is true, it raises **ValueError** on creating an index with duplicates.
* **sort:** It sorts the columns if the columns of self and other are not aligned. The default sorting is deprecated, and it will change to not-sorting in a future version of pandas. We pass **sort=True** Explicitly for silence the warning and the sort, whereas we pass **sort=False** Explicitly for silence the warning and not the sort.

1. **import** pandas as pd
2. info1 = pd.DataFrame({"x":[25,15,12,19], "y":[47, 24, 17, 29]})
3. Info2 = pd.DataFrame({"x":[25, 15, 12], "y":[47, 24, 17], "z":[38, 12, 45]})
4. info.append(info2, ignore\_index = True) //# append info2 at end in info1



1. info1 = info = pd.DataFrame({"x":[15, 25, 37, 42], "y":[24, 38, 18, 45]})
2. info2 = pd.DataFrame({"x":[15, 25, 37], "y":[24, 38, 45]})
3. print(info1, "\n")
4. print(info2)
5. info1.append(df2) # append info2 at the end of info1 dataframe
6. # Continuous index value will maintained
7. info.append(info2, ignore\_index = True) #across rows in **new** apended data fram



# Pandas DataFrame.apply()

The Pandas **apply()** function allows the user to pass a function and apply it to every single value of the Pandas series. This function improves the capabilities of the panda's library because it helps to segregate data according to the conditions required. So that it can be efficiently used for data science and machine learning.

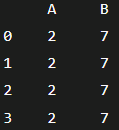
The objects that are to be passed to function are **Series objects** whose index is either the DataFrame's index, i.e., axis=0 or the DataFrame's columns, i.e., axis=1. By default, the **result\_type=None** and the final return type is inferred from the return type of the applied function. Otherwise, it depends on the **result\_type** argument.

**DataFrame.apply(func, axis=0, broadcast=None, raw=False, reduce=None, result\_type=None, args=(), \*\*kwds)**

* **func:** It is a function that is to be applied to each column or row.
* **axis:** {0 or 'index', 1 or 'columns'}, default value 0  
  It is an axis along which the function is applied. It can have two values:
  + 0 or 'index': It applies the function to each of the columns.
  + 1 or 'columns': It applies the function to each of the rows.
* **broadcast:** It is an optional parameter that returns the boolean values.  
  Only relevant for aggregation functions:  
  False or None: It returns a Series whose length will be the length of the index or the number of columns based on the axis parameter.  
  True: The results will be broadcasted to the original shape of the frame; the original index and columns will be retained.
* **raw:** bool, default value False  
  **False:** It passes each row or column as a Series to the function.  
  **True:** The passed function will receive a ndarray objects. If you are applying a NumPy reduction function, it will achieve better performance.
* **reduce:** bool or None, default value None  
  It tries to apply the reduction procedures. If the DataFrame is empty, the **apply** will use the **reduce** to determine whether the result should be a Series or a DataFrame.  
  By default, **reduce=None**, the **apply's** return value will be guessed by calling **func** on an empty Series (note: All the exceptions that are to be raised by func will be ignored while guessing). If **reduce=True**, Series will always be returned, whereas **reduce=False**, Always the DataFrame will be returned.
* **result\_type:** {'expand', 'reduce', 'broadcast', None}, default value None  
  These only act when axis=1 (columns):  
  **'expand':** It defines the list-like results that will be turned into columns.  
  **'reduce':** It is the opposite of '**expand**'. If possible, it returns a Series rather than expanding list-like results.  
  **'broadcast':** It broadcast the results to the original shape of the DataFrame, the original index, and the columns will be retained.  
  The default value **None** depends on the return value of the applied function , i.e., list-like results returned as a Series of those.  
  If **apply** returns a Series, it expands to the columns.
* **args:** It is a positional argument that is to be passed to **func** in addition to the array/series.
* **\*\*kwds:** It is an optional keyword argument, which is used to pass as keywords arguments to func.

**It returns the result of applying func along the given axis of the DataFrame.**

1. info = pd.DataFrame([[2, 7]] \* 4, columns=['P', 'Q'])
2. info.apply(np.sqrt)
3. info.apply(np.sum, axis=0)
4. info.apply(np.sum, axis=1)
5. info.apply(lambda x: [1, 2], axis=1)
6. info.apply(lambda x: [1, 2], axis=1, result\_type='expand')
7. info.apply(lambda x: pd.Series([1, 2], index=['foo', 'bar']), axis=1)
8. info.apply(lambda x: [1, 2], axis=1, result\_type='broadcast')
9. print(info)



# Pandas DataFrame.aggregate()

The main task of DataFrame.aggregate() function is to apply some aggregation to one or more column. Most frequently used aggregations are:

**sum:** It is used to return the sum of the values for the requested axis.

**min:** It is used to return the minimum of the values for the requested axis.

**max:** It is used to return the maximum values for the requested axis.

**DataFrame.aggregate(func, axis=0, \*args, \*\*kwargs)**

**func:** It refers callable, string, dictionary, or list of string/callables.

It is used for aggregating the data. For a function, it must either work when passed to a DataFrame or DataFrame.apply(). For a DataFrame, it can pass a dict, if the keys are the column names.

**axis: (default 0):** It refers to 0 or 'index', 1 or 'columns'

**0 or 'index':** It is an apply function for each column.

**1 or 'columns':** It is an apply function for each row.

**\*args:** It is a positional argument that is to be passed to **func**.

**\*\*kwargs:** It is a keyword argument that is to be passed to the **func**.

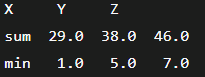
### Returns: It returns the scalar, Series or DataFrame.

**scalar:** It is being used when **Series.agg** is called with the single function.

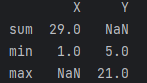
**Series:** It is being used when DataFrame.agg is called for the single function.

**DataFrame:** It is being used when DataFrame.agg is called for the several functions.

1. **import** pandas as pd
2. **import** numpy as np
3. info=pd.DataFrame([[1,5,7],[10,12,15],[18,21,24],[np.nan,np.nan,np.nan]],columns=['X','Y','Z'])
4. info.agg(['sum','min'])



1. **import** pandas as pd
2. **import** numpy as np
3. info=pd.DataFrame([[1,5,7],[10,12,15],[18,21,24],[np.nan,np.nan,np.nan]],columns=['X','Y','Z'])
4. df.agg({'X' : ['sum', 'min'], 'Y' : ['min', 'max']})



# Pandas DataFrame.assign()

The assign() method is also responsible for adding a new column into a DataFrame.

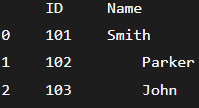
If we re-assign an existing column, then its value will be overwritten.

**DataFrame.assign(\*\*kwargs)**

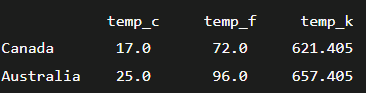
* **kwargs:** keywords are the column names. These keywords are assigned to the new column if the values are callable. If the values are not callable, they are simply assigned.

## Returns: It returns a new DataFrame with the addition of the new columns.

1. **import** pandas as pd
2. info = pd.DataFrame() # Create an empty dataframe
3. info['ID'] = [101, 102, 103] # Create a column
4. print(info) ## View the dataframe
5. # Assign a **new** column to dataframe called name
6. info.assign(Name = ['Smith', 'Parker', 'John'])



1. info = pd.DataFrame({'temp\_c': [17.0, 25.0]},
2. index=['Canada', 'Australia']) # Create an index that consist some values
3. info
4. info.assign(temp\_f=lambda x: x['temp\_c'] \* 6 / 2 + 21,
5. temp\_k=lambda x: (x['temp\_f'] + 342.27) \* 6 / 4)



# 

# Pandas DataFrame.astype()

The astype() method is generally used for casting the pandas object to a specified **dtype.astype()** function. It can also convert any suitable existing column to a categorical type.

It comes into use when we want to case a particular column data type to another data type. We can also use the input to Python dictionary to change more than one column type at once. In the dictionary, the key label corresponds to the column name, and the values label corresponds to the new data types that we want to be in the columns.

**DataFrame.astype(dtype, copy=True, errors='raise', \*\*kwargs)**

**dtype:** It uses numpy.dtype or the Python type for casting the entire pandas object to the same type. It can also use {col: dtype, ?} alternatively where col refers to the column label, and dtype is a numpy.dtype or Python type for casting one or more of the DataFrame's columns to column-specific types.

**copy:** If copy=True, it returns a copy. Be careful when setting copy= False because changes to values may propagate to other pandas objects.

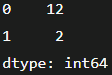
**errors:** For provided dtype, it controls the raising of exceptions on the invalid data.

* **raise:** It allows the exception that is to be raised.
* **ignore:** It ignores the exception. It returns the original object on error.

**kwargs:** It is a keyword argument that is to be passed on to the constructor.

### Returns: **casted:** It returns the same type as a caller.

1. **import** pandas as pd
2. a = {'col1': [1, 2], 'col2': [3, 4]}
3. info = pd.DataFrame(data=a)
4. info.dtypes
5. info.astype('int64').dtypes # We convert it into 'int64' type.
6. info.astype({'col1': 'int64'}).dtypes
7. x = pd.Series([1, 2], dtype='int64')
8. x.astype('category')
9. cat\_dtype = pd.api.types.CategoricalDtype(categories=[2, 1], ordered=True)
10. x.astype(cat\_dtype)
11. x1 = pd.Series([1,2])
12. x2 = x1.astype('int64', copy=False)
13. x2[0] = 10
14. x1 # note that x1[0] has changed too



# Pandas DataFrame.count()

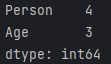
The Pandas count() is defined as a method that is used to count the number of non-NA cells for each column or row. It is also suitable to work with the non-floating data.

**DataFrame.count(axis=0, level=None, numeric\_only=False)**

* **axis:** *{0 or 'index', 1 or 'columns'}, default value 0*0 or 'index' is used for row-wise, whereas 1 or 'columns' is used for column-wise.
* **level:** *int or str.* It is an optional parameter. If an axis is hierarchical, it counts along with the particular level and collapsing into the DataFrame.
* **numeric\_only:** *bool, default value False.* It only includes int, float, or Boolean data.

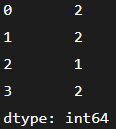
## Returns: It returns the count of Series or DataFrame if the level is specified.

1. **import** pandas as pd
2. **import** numpy as np
3. info = pd.DataFrame({"Person":["Parker", "Smith", "William", "John"],
4. "Age": [27., 29, np.nan, 32] })
5. info.count()



**Example 2:** If we want to count for each of the row, we can use the **axis** parameter.

1. info = pd.DataFrame({"Person":["Parker", "Smith", "William", "John"],
2. "Age": [27., 29, np.nan, 32] })
3. info.count(axis='columns')



# Pandas DataFrame.cut()

The **cut()** method is invoked when you need to segment and sort the data values into bins. It is used to convert a continuous variable to a categorical variable. It can also segregate an array of elements into separate bins. The method only works for the one-dimensional array-like objects.

If we have a large set of scalar data and perform some statistical analysis on it, we can use the **cut()** method.

**pandas.cut(x, bins, right=True, labels=None, retbins=False, precision=3, include\_lowest=False, duplicates='raise')**

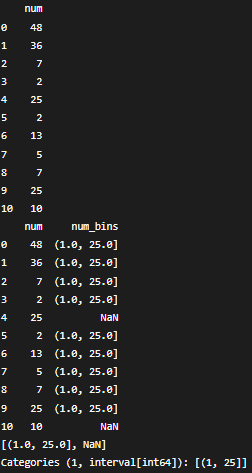
**x:** It generally refers to an array as an input that is to be bin. The array should be a one-dimensional array.

**bins:** It refers to an **int**, **sequence of scalars**, or **IntervalIndex** values that define the bin edges for segmentation. Most of the time, we have numerical data on a very large scale. So, we can group values into bins to easily perform descriptive statistics as a generalization of patterns in data. criteria for binning the data into groups are as folows:

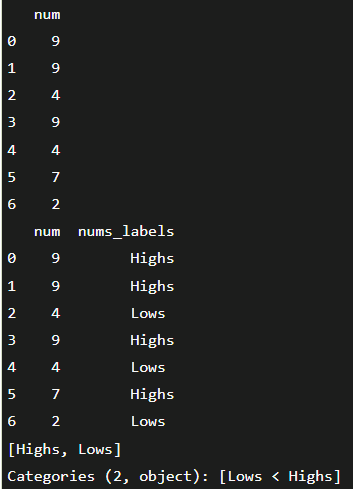
* **int:** It defines the number of equal-width bins that are in the range of **x**. We can also extend the range of **x** by **.1**% on both sides to include the minimum and maximum values of **x**.
* **sequence of scalars:** It mainly defines the bin edges that are allowed for non-uniform width.
* **IntervalIndex:** It refers to an exact bin that is to be used in the function. It should be noted that the **IntervalIndex** for bins must be non-overlapping.
* **right:** It consists of a boolean value that checks whether the **bins** include the rightmost edge or not. Its default value is True, and it is ignored when **bins** is an
* **labels:** It is an **optional** parameter that mainly refers to an array or a boolean value. Its main task is to specify the labels for the returned The length of the labels must be the same as the resulting bins. If we set its value to False, it returns only integer indicator of the bins. This argument is ignored if bins is an IntervalIndex.
* **retbins:** It refers to a boolean value that checks whether to return the bins or not. It is often useful when bins are provided as a scalar value. The default value of retbins is False.
* **precision:** It is used to store and display the bins labels. It consists of an integer value that has the default value **3**.
* **include\_lowest:** It consists of a boolean value that is used to check whether the first interval should be left-inclusive or not.
* **duplicates:** It is an **optional** parameter that decides whether to raise a ValueError or drop duplicate values if the bin edges are not unique.

## Returns: This method returns two objects as output which are as follows:

1. **out:** It mainly refers to a **Categorical**, **Series,** or **ndarray** that is an array-like object which represents the respective bin for each value of These objects depend on the value of **labels**. The possible values than can be returned are as follows:
   * **True:** It is a default value that returns a Series or a Categorical variable. The values stored in these objects are Interval data type.
   * **sequence of scalars:** It also returns a Series or a Categorical variable. The values that are stored in these objects are the type of the sequence.
   * **False:** The false value returns an ndarray of integers.
2. **bins:** It mainly refers to a **ndarray**
3. **import** pandas as pd
4. **import** numpy as np
5. info\_nums = pd.DataFrame({'num': np.random.randint(1, 50, 11)})
6. print(info\_nums)
7. info\_nums['num\_bins'] = pd.cut(x=df\_nums['num'], bins=[1, 25, 50])
8. print(info\_nums)
9. print(info\_nums['num\_bins'].unique())



1. info\_nums = pd.DataFrame({'num': np.random.randint(1, 10, 7)})
2. print(info\_nums)
3. info\_nums['nums\_labels'] = pd.cut(x=info\_nums['num'], bins=[1, 7, 10], labels=['Lows', 'Highs'], right=False)
4. print(info\_nums)
5. print(info\_nums['nums\_labels'].unique())



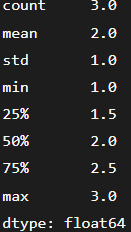
# Pandas DataFrame.describe()

The describe() method is used for calculating some statistical data like **percentile, mean** and **std** of the numerical values of the Series or DataFrame. It analyzes both numeric and object series and also the DataFrame column sets of mixed data types.

**DataFrame.describe(percentiles=None, include=None, exclude=None)**

### Returns: It returns the statistical summary of the Series and DataFrame.

1. **import** pandas as pd
2. **import** numpy as np
3. a1 = pd.Series([1, 2, 3])
4. a1.describe()



1. info = pd.DataFrame({'categorical': pd.Categorical(['s','t','u']), 'numeric': [1, 2, 3],
2. 'object': ['p', 'q', 'r'] })
3. info.describe()
4. info.describe(include='all')
5. info.numeric.describe()
6. info.describe(include=[np.number])
7. info.describe(include=[np.object])
8. info.describe(include=['category'])
9. info.describe(exclude=[np.number])
10. info.describe(exclude=[np.object])



# Pandas DataFrame.drop\_duplicates()

drop\_duplicates() funct performs common data cleaning task that deals with duplicate values in DataFrame. This method helps in removing duplicate values from DataFrame.

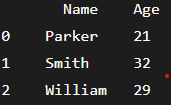
**DataFrame.drop\_duplicates(subset=None, keep='first', inplace=False)**

* **subset:** It takes a column or the list of column labels. It considers only certain columns for identifying duplicates. Default value **None**.
* **keep:** It is used to control how to consider duplicate values. It has three distinct values that are as follows:
  + **first:** It drops the duplicate values except for the first occurrence.
  + **last:** It drops the duplicate values except for the last occurrence.
  + **False:** It drops all the duplicates.
* **inplace:** Returns the boolean value. Default value is False.

**If it is true, it removes the rows with duplicate values.**

### Return: Depending on the arguments passed, it returns the DataFrame with the removal of duplicate rows.

1. **import** pandas as pd
2. emp = {"Name": ["Parker", "Smith", "William", "Parker"], "Age": [21, 32, 29, 21]}
3. info = pd.DataFrame(emp)
4. info = info.drop\_duplicates()
5. print(info)



# Pandas DataFrame.groupby()

In Pandas, **groupby()** function allows us to rearrange data by utilizing them on real-world data sets. Its primary task is to split the data into various groups. These groups are categorized based on some criteria. The objects can be divided from any of their axes.

**DataFrame.groupby(by=None, axis=0, level=None, as\_index=True, sort=True, group\_keys=True, squeeze=False, \*\*kwargs)**

This operation consists of the following steps for aggregating/grouping the data:

* **Splitting datasets**
* **Analyzing data**
* **Aggregating or combining data**

#### Note: The result of Groupby operation is not a DataFrame, but dict of DataFrame objects.

## Split data into groups: There are multiple ways to split any object into group.like..

* obj.groupby('key')
* obj.groupby(['key1','key2'])
* obj.groupby(key,axis=1)

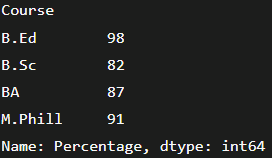
We can also add some functionality to each subset. The following operations can be performed on the applied functionality:

* **Aggregation:** Computes summary statistic.
* **Transformation:** It performs some group-specific operation.
* **Filtration:** It filters the data by discarding it with some condition.

### Aggregations

It is defined as a function that returns a single aggregated value for each of the groups. We can perform several aggregation operations on the grouped data when the **groupby** object is created.

1. data = {'Name': ['Parker', 'Smith', 'John', 'William'], 'Percentage': [82, 98, 91, 87],
2. 'Course': ['B.Sc','B.Ed','M.Phill','BA']}
3. df = pd.DataFrame(data)
4. grouped = df.groupby('Course')
5. print(grouped['Percentage'].agg(np.mean))



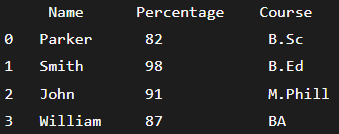
### Transformations

It is an operation on a group or column that performs some group-specific computation and returns an object that is indexed with the same size as of the group size.

1. data = {'Name': ['Parker', 'Smith', 'John', 'William'], ‘'Percentage': [82, 98, 91, 87],
2. 'Course': ['B.Sc','B.Ed','M.Phill','BA']}
3. df = pd.DataFrame(data)
4. grouped = df.groupby('Course')
5. Percentage = lambda x: (x - x.mean()) / x.std()\*10
6. print(grouped.transform(Percentage))

### Filtration: The **filter()** function filters the data by defining some criteria and returns the subset of data.

1. data = {'Name': ['Parker', 'Smith', 'John', 'William'], 'Percentage': [82, 98, 91, 87],
2. 'Course': ['B.Sc','B.Ed','M.Phill','BA']}
3. df = pd.DataFrame(data)
4. grouped = df.groupby('Course')
5. print (df.groupby('Course').filter(lambda x: len(x) >= 1))



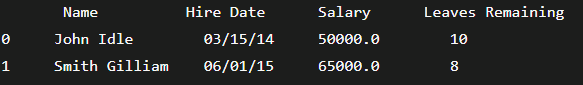
# Pandas DataFrame.head()

The head() returns the first n rows for the object based on position. If your object has the right type of data in it, it is useful for quick testing. This method is used for returning top n (by default value 5) rows of a data frame or series.

**DataFrame.head(n=5)**

### Return : It returns the DataFrame with top n rows.

1. **import** pandas as pd
2. data = pd.read\_csv("aa.csv") # making data frame
3. data\_top = data.head(2) # calling head() method storing in **new** variable
4. data\_top



# Pandas DataFrame.hist()

# The hist() function is defined as a quick way to understand the distribution of certain numerical variables from the dataset. It divides the values within a numerical variable

into "**bins**". It counts the number of examinations that fall into each of the bin. These bins are responsible for a rapid and intuitive sense of the distribution of the values within a variable by visualizing bins.

We can create a histogram by using the **DataFrame.hist()** method, which is a wrapper for the matplotlib pyplot API.

It is also a useful tool that quickly access the probability distribution.

**DataFrame.hist(data, column=None, by=None, grid=True, xlabelsize=None, xrot=None, ylabelsize=None, yrot=None, ax=None, sharex=False, sharey=False, figsize=None, layout=None, bins=10, \*\*kwds)**

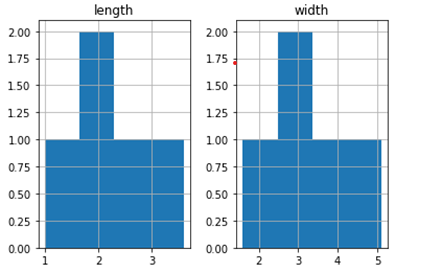
* **data:** A DataFrame.It is a pandas DataFrame object that holds the data.
* **column:** Refers to a string or sequence. If it is passed, it will be used to limit the data to a subset of columns.
* **by:** It is an optional parameter. If it is passed, then it will be used to form the histogram for independent groups.
* **grid:** It is an optional parameter. Used for showing axis grid lines. Default is True.
* **xlabelsize:** Refers to the integer value. Default value None. Used for specifying the changes in the x-axis label size.
* **xrot:** Refers to float value. Used for rotating the x-axis labels. Default value None.
* **ylabelsize:** Refers to an integer value. Used for specifying the changes in the y-axis label size.
* **yrot:** Refers to the float value. Used for rotating y-axis labels. Default value None.
* **ax:** Matplotlib axes object.  
  It defines the axis on which we need to plot the histogram. Default value None.
* **sharex:** Refers to the boolean value. Default value True, if ax is None else False. In the case of subplots, if value is True, it shares the x-axis and sets some of the x-axis labels to invisible. Its Default value is True.  
  If the ax is none, it returns False if an ax is passed in.

#### **Note:** Passing true in both an ax and sharex, it will alter all x-axis labels for all the subplots

* **sharey:** Default value False. In the case of subplots is True, it shares the y-axis and sets some y-axis labels to invisible.
* **figsize:** Refers to the size in inches for the figure to create. By default, it uses the value in **matplotlib.rcParams**.
* **layout:** It is an optional parameter. It returns the tuple of (rows, columns) for the layout of the histograms.
* **bins:** Default value 10. It refers to number of histogram bins that are to be used. If an integer value is given, then it returns calculated value of bins +1 bin edges.
* **\*\*kwds:** Refers to all the other plotting keyword arguments that are to be passed to matplotlib.pyplot.hist().

### Returns: It returns the matplotlib.AxesSubplot or numpy.ndarray.

1. info = pd.DataFrame({
2. 'length': [2, 1.7, 3.6, 2.4, 1], 'width': [4.2, 2.6, 1.6, 5.1, 2.9] })
3. hist = info.hist(bins=4)

\

# Pandas DataFrame.iterrows()

If you want to loop over the DataFrame for performing some operations on each of the rows then you can use iterrows() function in Pandas.Pandas use three functions for iterating over the rows of the DataFrame, i.e., **iterrows**(), **iteritems**() and **itertuples**().

## Iterate rows with Pandas iterrows:

The iterrows () is responsible for loop through each row of the DataFrame. It returns an iterator that contains index and data of each row as a Series.

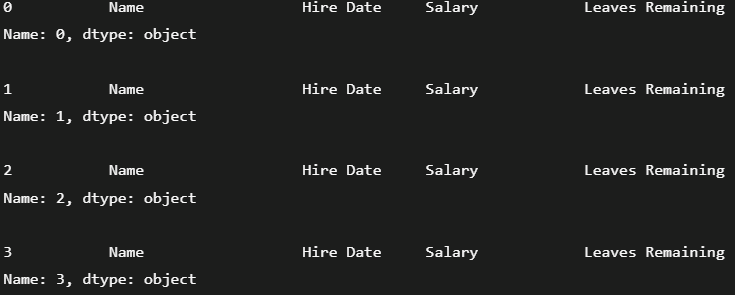
We have the next function to see the content of the iterator.

This function returns each index value along with a series that contain data in each row.

* **iterrows()** - used for iterating over the rows as (index, series) pairs.
* **iteritems()** - used for iterating over the (key, value) pairs.
* **itertuples()** - used for iterating over the rows as namedtuples.

### Yields:

1. **index:** Returns the index of the row and a tuple for the MultiIndex.
2. **data:** Returns the data of the row as a Series.
3. **it:** Returns a generator that iterates over the rows of the frame.
   1. data = pd.read\_csv("aa.csv") # making data frame from csv file
   2. **for** i, j in data.iterrows():
   3. print(i, j)



# Pandas DataFrame.join()

The method of combining the DataFrame using common fields is called "**joining**". The method that we use for combining the DataFrame is a **join()** method. The columns that contain common values are called "**join key"**.

The join() method is often useful when one DataFrame is a lookup table that contains additional data added into the other DataFrame. It is a convenient method that can combine the columns of two differently-indexed DataFrames into a single DataFrame.

## Identifying join keys

To determine the appropriate join keys, first, we have to define required fields that are shared between the DataFrames. Both the DataFrames consist of the columns that have the same name and also contain the same data.

## Inner joins: Basically, its main task is to combine the two DataFrames based on a join key and returns a new DataFrame. The returned DataFrame consists of only selected rows that have matching values in both of the original DataFrame.

## Left joins: If we want to add some information into the DataFrame without losing any of the data, we can simply do it through a different type of join called a "**left outer join**"..

Like an inner join, left join also uses the join keys to combine two DataFrames, but unlike inner join, it returns all of the rows from the left DataFrame, even those rows whose join keys do not include the values in the right DataFrame.

## Syntax: **DataFrame.join(other, on=None, how='left', lsuffix='', rsuffix='', sort=False)**

## Parameters: **other:** It refers to the DataFrame or Series.

In this case, index should be similar to one of the columns. If we pass a Series, named attribute has to be set for using it as the column name in resulting joined DataFrame.

**on:** It is an optional parameter that refers to **array-like** or **str** values.

It refers to a column or index level name in the caller to join on the index. Otherwise, it joins index-on-index. If multiple values are present, then the **other** DataFrame must have MultiIndex. It is like an Excel VLOOKUP operation that can pass an array as the join key if it is not already contained within the calling DataFrame.

**how:** It refers to 'left', 'right', 'outer', 'inner' values that mainly work on how to handle the operation of the two objects. The default value of **how** is **left.**

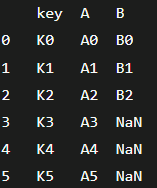
* **left:** It uses a calling frame's index or column if the parameter on is specified.
* **right:** It uses the other index.
* **outer:** It is used to form a union of calling frame's index or column if parameter on is specified with other's index, and also sort it lexicographically.
* **inner:** It is used to form an intersection of calling frame's index or column if parameter **on** is specified with other's index. So, due to this, it preserves the order of the calling object.

**lsuffix:** It refers to a string object that has the default value ''. It uses the Suffix from the left frame's overlapping columns.

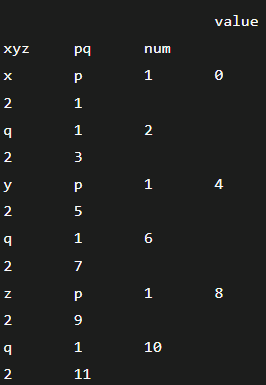
**rsuffix:** It refers to a string value, that has the default value ''. It uses the Suffix from the right frame's overlapping columns.

**sort:** It consists of a boolean value that sorts the resulting DataFrame lexicographically by the join key. If we pass False value, then the order of the join key mainly depends on the join type, i.e., **how**.

1. **import** pandas as pd
2. info = pd.DataFrame({'key': ['K0', 'K1', 'K2', 'K3', 'K4', 'K5'],
3. 'A': ['A0', 'A1', 'A2', 'A3', 'A4', 'A5']})
4. x = pd.DataFrame({'key': ['K0', 'K1', 'K2'],
5. 'B': ['B0', 'B1', 'B2']})
6. info.join(x, lsuffix='\_caller', rsuffix='\_x')
7. info.set\_index('key').join(x.set\_index('key'))
8. info.join(x.set\_index('key'), on='key')



1. **import** pandas as pd
2. leftindex = pd.MultiIndex.from\_product([list('xyz'), list('pq'), [1, 2]],
3. names=['xyz', 'pq', 'num'])
4. left = pd.DataFrame({'value': range(12)}, index=leftindex)
5. left



Pandas DataFrame.mean()

The mean() function is used to return the mean of the values for the requested axis. If we apply this method on a **Series object**, then it returns a **scalar value**, which is the mean value of all the observations in the dataframe.

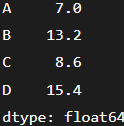
If we apply this method on a DataFrame object, then it returns a Series object which contains mean of values over the specified axis.

### Syntax:**DataFrame.mean(axis=None, skipna=None, level=None, numeric\_only=None, \*\*kwargs)**

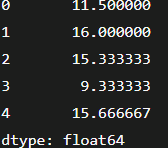
* **axis:** {index (0), columns (1)}.This refers to axis for a functn that is to be applied.
* **skipna:** It excludes all the null values when computing result.
* **level:** It counts along with a particular level and collapsing into a Series if the axis is a MultiIndex (hierarchical),
* **numeric\_only:** It includes only int, float, boolean columns. If None, it will attempt to use everything, then use only numeric data. Not implemented for Series.

### Returns: It returns the mean of the Series or DataFrame if the level is specified.

1. **import** pandas as pd
2. info = pd.DataFrame({"A":[8, 2, 7, 12, 6], "B":[26, 19, 7, 5, 9], "C":[10, 11, 15, 4, 3],
3. "D":[16, 24, 14, 22, 1]})
4. # If axis = 0 is not specified, then by **default** method **return** mean over index axis
5. info.mean(axis = 0)



1. info = pd.DataFrame({"A":[5, 2, 6, 4, None], "B":[12, 19, None, 8, 21],
2. "C":[15, 26, 11, None, 3], "D":[14, 17, 29, 16, 23]})
3. info.mean(axis = 1, skipna = True) # **while** finding mean, it skip **null** values



# Pandas melt()

Pandas.melt() function is used to **unpivot** DataFrame from wide format to long format.

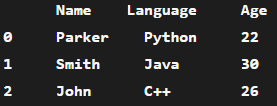
Its main task is to massage a DataFrame into a format where some columns are identifier variables and remaining columns are considered as measured variables, are unpivoted to the row axis. It leaves just two non-identifier columns, variable and value.

### Syntax:**pandas.melt(frame, id\_vars=None, value\_vars=None, var\_name=None, value\_name='value', col\_level=None)**

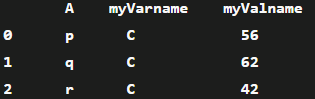
* **frame:** It refers to the DataFrame.
* **id\_vars[tuple, list, or ndarray, optional]:** It refers to the columns to use as identifier variables.
* **value\_vars[tuple, list, or ndarray, optional]:** Refers to columns to unpivot. If it is not specified, use all columns that are not set as id\_vars.
* **var\_name[scalar]:** Refers to a name to use for the 'variable' column. If it is None, it uses frame.columns.name or 'variable'.
* **value\_name[scalar, default 'value']:** Refers to a name to use for 'value' column.
* **col\_level[int or string, optional]:** It’ll use this level to melt if colms are MultiIndex.

### Returns:**It returns the unpivoted DataFrame as the output.**

1. **import pandas as pd**
2. **info = pd.DataFrame({'Name': {0: 'Parker', 1: 'Smith', 2: 'John'},**
3. **'Language': {0: 'Python', 1: 'Java', 2: 'C++'}, 'Age': {0: 22, 1: 30, 2: 26}})**
4. **# Name is id\_vars and Course is value\_vars**
5. **pd.melt(info, id\_vars =['Name'], value\_vars =['Language'])**
6. **info**



1. **info = pd.DataFrame({'A': {0: 'p', 1: 'q', 2: 'r'}, 'B': {0: 40, 1: 55, 2: 25},**
2. **'C': {0: 56, 1: 62, 2: 42}})**
3. **pd.melt(info, id\_vars=['A'], value\_vars=['C'])**
4. **pd.melt(info, id\_vars=['A'], value\_vars=['B', 'C'])**
5. **pd.melt(info, id\_vars=['A'], value\_vars=['C'], var\_name='myVarname', value\_name='myValname')**



# Pandas DataFrame.merge()

Pandas **merge()** is defined as the process of bringing the two datasets together into one and aligning the rows based on the common attributes or columns. It is an entry point for all standard database join operations between DataFrame objects:

## Syntax:**pd.merge(left, right, how='inner', on=None, left\_on=None, right\_on=None,**

**left\_index=False, right\_index=False, sort=True)**

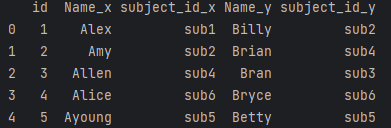
* **right:** *DataFrame or named Series.* It is an object which merges with DataFrame.
* **how:** *{'left', 'right', 'outer', 'inner'}, default 'inner'* Type of merge to be performed.
  + **left:** It use only keys from the left frame, similar to a SQL left outer join; preserve key order.
  + **right:** It use only keys from the right frame, similar to a SQL right outer join; preserve key order.
  + **outer:** It used the union of keys from both frames, similar to a SQL full outer join; sort keys lexicographically.
  + **inner:** It use the intersection of keys from both frames, similar to a SQL inner join; preserve the order of the left keys.
* **on:** *label or list.* It is a column or index level names to join on. It must be found in both the left and right DataFrames. If on is None and not merging on indexes, then this defaults to the intersection of the columns in both DataFrames.  
  **left\_on:** *label or list, or array-like.* It is a column or index level names from the left DataFrame to use as a key. It can be an array with length equal to the length of the DataFrame.
* **right\_on:** *label or list, or array-like.* It is a column or index level names from the right DataFrame to use as keys. It can be an array with length equal to the length of the DataFrame.
* **left\_index :** *bool, default False.* It uses the index from left DataFrame as the join key(s), If true. In case of MultiIndex (hierarchical), many keys in other DataFrame (either the index or some columns) should match the number of levels.
* **right\_index :** *bool, default False.* It uses the index from the right DataFrame as the join key. It has the same usage as the left\_index.
* **sort:** *bool, default False.* If True, it sorts the join keys in lexicographical order in the result DataFrame. Otherwise, the order of the join keys depends on the join type (how keyword).
* **suffixes:** *tuple of the (str, str), default ('\_x', '\_y').* It suffixes to apply to overlap the column names in the left and right DataFrame, respectively. The columns use (False, False) values to raise an exception on overlapping.
* **copy:** *bool, default True.* If True, it returns a copy of the DataFrame.Otherwise, It can avoid the copy.
* **indicator:** *bool or str, default False.* If True, It adds a column to output DataFrame "**\_merge**" with information on the source of each row. If it is a string, a column with information on the source of each row will be added to output DataFrame, and the column will be named value of a string. The information column is defined as a categorical-type and it takes value of:
  + **"left\_only"** for the observations whose merge key appears only in 'left' of the DataFrame, whereas,
  + **"right\_only"** is defined for observations in which merge key appears only in 'right' of the DataFrame,
  + **"both"** if the observation's merge key is found in both of them.
* **validate:** *str, optional,* If it is specified, it checks merge type that is given below:
  + "one\_to\_one" or "1:1": It checks if merge keys are unique in both the left and right datasets.
  + "one\_to\_many" or "1:m": It checks if merge keys are unique in only the left dataset.
  + "many\_to\_one" or "m:1": It checks if merge keys are unique in only the right dataset.
  + "many\_to\_many" or "m:m": It is allowed, but does not result in checks.

### Example1: Merge two DataFrames on a key

1. **import** pandas as pd
2. left = pd.DataFrame({ 'id':[1,2,3,4], 'Name': ['John', 'Parker', 'Smith', 'Parker'],
3. 'subject\_id':['sub1','sub2','sub4','sub6']})
4. right = pd.DataFrame({ 'id':[1,2,3,4], 'Name': ['William', 'Albert', 'Tony', 'Allen'],
5. 'subject\_id':['sub2','sub4','sub3','sub6']})
6. print (left)
7. print (right)

### Example2: Merge two DataFrames on multiple keys:

1. left = pd.DataFrame({ 'id':[1,2,3,4,5], 'Name': ['Alex', 'Amy', 'Allen', 'Alice', 'Ayoung'],
2. 'subject\_id':['sub1','sub2','sub4','sub6','sub5']})
3. right = pd.DataFrame({ 'id':[1,2,3,4,5], 'Name': ['Billy', 'Brian', 'Bran', 'Bryce', 'Betty'],
4. 'subject\_id':['sub2','sub4','sub3','sub6','sub5']})
5. print pd.merge(left,right,on='id')



# Pandas DataFrame.pivot\_table()

The Pandas **pivot\_table()** is used to calculate, aggregate, and summarize your data. It is defined as a powerful tool that aggregates data with calculations such as **Sum, Count, Average, Max,** and **Min**.

It also allows the user to sort and filter your data when the pivot table has been created.

### Parameters: **data:** A DataFrame.

* **values:** It is an **optional** parameter and refers the column to aggregate.
* **index:** It refers to the column, Grouper, and array.

If we pass an array, it must be of the same length as data.

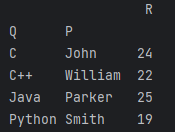
* **columns:** Refers to column, Grouper, and array

If we pass an array, it must be of the same length as data.

* **aggfunc:** function, list of functions, dict, default numpy.mean If we pass the list of functions, the resulting pivot table will have hierarchical columns whose top level are the function names.  
  If we pass a dict, the key is referred to as a column to aggregate, and value is function or list of functions.
* **fill\_value[scalar, default None]:** It replaces the missing values with a value.
* **margins[boolean, default False]:** It add all the row / columns (e.g. for subtotal / grand totals)
* **dropna[boolean, default True] :** It drops the columns whose entries are all NaN.
* **margins\_name[string, default 'All'] :** It refers to the name of the row/column that will contain the totals when margins are True.

### Returns: It returns a DataFrame as the output.

1. **import** pandas as pd
2. info = pd.DataFrame({'P': ['Smith', 'John', 'William', 'Parker'], 'Q': ['Python', 'C', 'C++', 'Java'], 'R': [19, 24, 22, 25]})
3. table = pd.pivot\_table(info, index =['P', 'Q'])
4. table



# Pandas DataFrame.query()

For analyzing the data, we need a lot of filtering operations. Pandas provide a query() method to filter the DataFrame.

It offers a simple way of making the selection and also capable of simplifying the task of index-based selection.

### Syntax: **DataFrame.query(expr, inplace=False, \*\*kwargs)**

* **expr:** Refers to an expression in string form to filter data.
* **inplace:** If the value is True, it makes the changes in the original DataFrame.
* **kwargs:** Refers to the other keyword arguments.

### Return: It returns a DataFrame that results from the query expression.

#### **Note:** This method only works if the column name doesn't have any empty spaces. You can replace the spaces in column names with '\_'

1. info = pd.DataFrame({'X': range(1, 6), 'Y': range(10, 0, -2), 'Z Z': range(10, 5, -1)})
2. info.query('X > Y')
3. info[info.X > info.Y]
4. info[info.Y == info['Z Z']]



# Pandas DataFrame.rename()

The main task of the Pandas **rename()** function is to **rename any index, column, or row**. This method is useful for renaming some selected columns because we have to specify the information only for those columns that we want to rename.

It mainly alters the axes labels based on some of the mapping (dict or Series) or the arbitrary function. The function must be unique and should range from **1** to **-1**. The labels will be left, if it is not contained in a dict or Series. If you list some extra labels, it will throw an error.

## Syntax: **DataFrame.rename(mapper=None, index=None, columns=None, axis=None, copy=True, inplace=False, level=None, errors='ign**

* **mapper:** It is a **dict-like** or **function** transformation that is to be applied to a particular axis label. We can use either **mapper** or **axis** to specify the axis targeted with **mapper, index**, and
* **index:** It is an alternative of specifying the axis (mapper, axis =0 is equivalent to the **index=mapper**).
* **columns:** It is an alternative to specify an axis (mapper, axis =1 is equivalent to the **columns=mapper**).
* **axis:** It refers to an **int** or **str** value that defines the axis targeted with the **mapper**. It can be either the axis name ('index', 'columns') or the number.
* **copy:** It refers to a boolean value that copies the underlying data. The default value of the **copy** is True.
* **inplace:** It refers to a boolean value and checks whether to return the new DataFrame or not. If it is true, it makes the changes in the original DataFrame. The default value of the **inplace** is True.
* **level:** It refers to an **int** or **level name** values that specify the level, if DataFrame has a multiple level index. The default value of the **level** is None.
* **errors:** It refers to **ignore, raise** If we specify **raise** value, it raises a **KeyError** if any of the labels are not found in the selected axis.

## Returns: It returns the DataFrame with renamed axis labels.

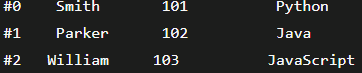
1. **import** pandas as pd
2. info = {'name': ['Parker', 'Smith', 'William', 'Robert'], 'age': [38, 47, 44, 34],
3. 'language': ['Java', 'Python', 'JavaScript', 'Python']}
4. info\_pd = pd.DataFrame(info)
5. info\_pd.rename(columns = {'name':'Name'}, inplace = True)
6. print("\nAfter modifying first column:\n", info\_pd.columns) # After renaming



1. info = {'name': ['Parker', 'Smith', 'William', 'Robert'], 'age': [38, 47, 44, 34],
2. 'language': ['Java', 'Python', 'JavaScript', 'Python']}
3. info\_pd = pd.DataFrame(info)
4. info\_pd.rename(columns = {'name':'Name', 'age':'Age', 'language':'Language'}, inplace = True)
5. print(info\_pd.columns) # After renaming columns



1. **import** pandas as pd
2. data = {'Name': ['Smith', 'Parker', 'William'], 'Emp\_ID': [101, 102, 103], 'Language': ['Python', 'Java', 'JavaScript']}
3. info1 = pd.DataFrame(data)
4. info2 = info.rename(index={0: '#0', 1: '#1', 2: '#2'})
5. print('Renamed Indexes:\n', info2)



# Pandas Dataframe.sample()

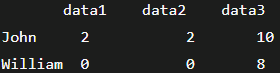
The Pandas sample() is used to select the rows and columns from the DataFrame randomly. If we want to build a model from an extensive dataset, we have to randomly choose a smaller sample of the data that is done through a function **sample**.

### Syntax: **DataFrame.sample(n=None, frac=None, replace=False, weights=None, random\_state=None, axis=None)**

* **n:** It is an optional parameter that consists of an integer value and defines the number of random rows generated.
* **frac:** It is also an optional parameter that consists of float values and returns **float value \* length of data frame values**. It cannot be used with a parameter n.
* **replace:** It consists of boolean value. If it is true, it returns a sample with replacement. The default value of the replace is false.
* **weights:** It is also an **optional** parameter that consists of str or ndarray-like. Default value "**None**" that results in equal probability weighting. If a DataFrame is being passed when **axis =0;** it will accept the name of a column.
* **random\_state:** It is also an **optional** parameter that consists of an integer or numpy.random.RandomState. If the value is int, it seeds for the random number generator or numpy RandomState object.
* **axis:** It is also an optional parameter that consists of integer or string value. 0 or '**row**' and 1 or 'column'.

### Returns: It returns a new object of the same type as a caller that contains n items randomly sampled from the caller object.

1. **import** pandas as pd
2. info = pd.DataFrame({'data1': [2, 4, 8, 0], 'data2': [2, 0, 0, 0], 'data3': [10, 2, 1, 8]},
3. index=['John', 'Parker', 'Smith', 'William'])
4. info['data1'].sample(n=3, random\_state=1)
5. info.sample(frac=0.5, replace=True, random\_state=1)
6. info.sample(n=2, weights='data3', random\_state=1)



**Ex:I**n this example, we take a csv file and extract random rows from the DataFrame by using a sample.

1. **import** pandas as pd
2. data = pd.read\_csv("aa.csv")
3. row1 = data.sample(n = 1) # randomly select one row
4. print(row1) # display row
5. row2 = data.sample(n = 2) # randomly select another row

# Pandas DataFrame.shift()

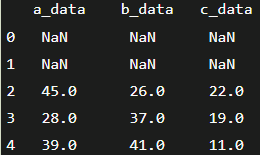
If you want to shift your column or subtract the column value with the previous row value from the DataFrame, you can do it by using the **shift()** function. It consists of a scalar parameter called **period**, which is responsible for showing the number of shifts to be made over the desired axis. It is also capable of dealing with time-series data.

## Syntax: **DataFrame.shift(periods=1, freq=None, axis=0)**

* **periods:** It consists of an integer value that can be positive or negative. It defines the number of periods to move.
* **freq:** It can be used with DateOffset, tseries module, str or time rule (e.g., 'EOM').
* **axis:** 0 is used for shifting the index, whereas 1 is used for shifting the column.
* **fill\_value:** Used for filling newly missing values.

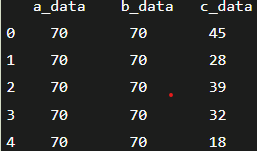
## Returns: It returns a shifted copy of DataFrame.

1. **import** pandas as pd
2. info= pd.DataFrame({'a\_data': [45, 28, 39, 32, 18], 'b\_data': [26, 37, 41, 35, 45],
3. 'c\_data': [22, 19, 11, 25, 16]})
4. info.shift(periods=2)



**Exm2:** This shows how to fill the missing values in the DataFrame using the **fill\_value**.

1. info= pd.DataFrame({'a\_data': [45, 28, 39, 32, 18], 'b\_data': [26, 38, 41, 35, 45],
2. 'c\_data': [22, 19, 11, 25, 16]})
3. info.shift(periods=2)
4. info.shift(periods=2,axis=1,fill\_value= 70) # axis=0 for row and 1 for col



# Pandas DataFrame.sort()

We can efficiently perform sorting in the DataFrame through different kinds:

* **By label**
* **By Actual value**

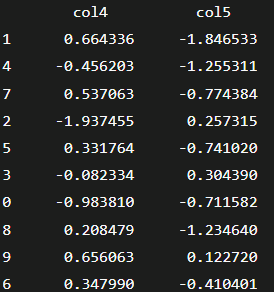
**By label:**The DataFrame can be sorted by using the **sort\_index()** method. It can be done by passing the axis arguments and the order of sorting. The sorting is done on row labels in ascending order by default.

1. info=pd.DataFrame(np.random.randn(10,2),index=[1,2,5,4,8,7,9,3,0,6],columns = ['col4','col3'])
2. info2=info.sort\_index()



* **Order of Sorting:** The order of sorting can be controlled by passing the Boolean value to the ascending parameter.

1. info= pd.DataFrame(np.random.randn(10,2),index=[1,4,7,2,5,3,0,8,9,6],columns = ['col4','col5'])
2. info\_2 = info.sort\_index(ascending=False)



* **Sort the Columns:** We can sort the columns labels by passing the axis argument respected to its values 0 or 1. By default, the **axis=0**, it sort by row.

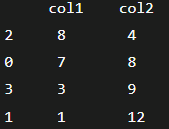
1. info = pd.DataFrame(np.random.randn(10,2),index=[1,4,8,2,0,6,7,5,3,9],columns = ['col4','col7'])
2. info\_2=info.sort\_index(axis=1)

## By Actual Value

It is another kind through which sorting can be performed in the DataFrame. Like index sorting, **sort\_values()** is a method for sorting by the values.

It also provides a feature in which we can specify the column name of the DataFrame with which values are to be sorted. It is done by passing the '**by**' argument.

1. info = pd.DataFrame({'col1':[7,1,8,3],'col2':[8,12,4,9]})
2. info\_2 = info.sort\_values(by='col2')



In the above output, observe that the values are sorted in **col2** only, and the respective **col1** value and row index will alter along with **col2**. Thus, they look unsorted.

* **columns:** Before Sorting, you have to pass an object or the column names.
* **ascending:** A Boolean value is passed that is responsible for sorting in the ascending order. Its default value is True.
* **axis:** 0 or index; 1 or 'columns'. The default value is 0. It decides whether you sort by index or columns.
* **inplace:** A Boolean value is passed. default value is false. It will modify any other views on this object & does not create a new instance while sorting DataFrame.
* **kind:** *'heapsort', 'mergesort', 'quicksort'*. It is an optional parameter that is to be applied only when you sort a single column or labels.
* **na\_position:** *'first', 'last'*. The *'first'* puts NaNs at the beginning, while the 'last' puts NaNs at the end. Default option last.

# Pandas DataFrame.sum()

Pandas **DataFrame.sum()** function is used to return the sum of the values for the requested axis by the user. If the input value is an index axis, then it will add all the values in a column and works same for all the columns. It returns a series that contains the sum of all the values in each column.

It is also capable of skipping the missing values in the DataFrame while calculating the sum in the DataFrame.

### Syntax: **DataFrame.sum(axis=None, skipna=None, level=None, numeric\_only=None, min\_count=0, \*\*kwargs)**

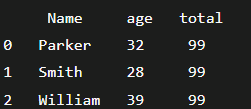
* **axis:** {index (0), columns (1)} 0 or 'index' is used for row-wise, whereas 1 or 'columns' is used for column-wise.
* **skipna:** bool, default True. It is used to exclude all the null values.
* **level:** int or level name, default None. It counts along a particular level and collapsing into a series, if the axis is a multiindex.
* **numeric\_only:** bool, default value None. It includes only int, float, & boolean columns. If it is None, it will attmpt to use everything, so num data must be used.
* **min\_count:** int, default value 0. It refers to the required number of valid values to perform any operation. If it is fewer than the **min\_count** non-NA values are present, then the result will be NaN.
* **\*\*kwargs:** It is an optional parameter that is to be passed to a function.

### Returns: It returns the sum of Series or DataFrame if a level is specified.

1. pd.Series([]).sum() #0 # **default** min\_count = 0
2. # Passed min\_count = 1, then sum of an empty series will be NaN
3. pd.Series([]).sum(min\_count = 1) #NaN

Ex. info = {'Name': ['Parker', 'Smith', 'William'], 'age' : [32, 28, 39]}

1. data = pd.DataFrame(info)
2. data['total'] = data['age'].sum() # sum of all salary stored in 'total'
3. print(data)



# Pandas DataFrame.to\_excel()

We can export the DataFrame to the excel file by using the to\_excel() function.

To write a single object to the excel file, we have to specify the target file name. If we want to write to multiple sheets, we need to create an **ExcelWriter** object with target filename and also need to specify the sheet in the file in which we have to write.

The multiple sheets can also be written by specifying the unique **sheet\_name**. It is necessary to save the changes for all the data written to the file.

#### **Note:** If we create an ExcelWriter object with a file name that already exists, it will erase the content of the existing file.

### Syntax: **DataFrame.to\_excel(excel\_writer, sheet\_name='Sheet1', na\_rep='', float\_format=None, columns=None, header=True, index=True, index\_label=None, startrow=0, startcol=0, engine=None, merge\_cells=True, encoding=None, inf\_rep='inf', verbose=True, freeze\_panes=None)**

* **excel\_writer:** A file path or existing ExcelWriter.
* **sheet\_name:** It refers to the name of the sheet that contains the DataFrame.
* **na\_repr:** Missing Data representation.
* **float\_format:** It is an optional parameter that formats the string for floating-point numbers.
* **columns:** Refers the column to write.
* **header:** It writes out the column names. If a list of the string is given, it is assumed to be the aliases for the column names.
* **index:** It writes the index.
* **index\_label:** Refers to the column label for the index column. If it is not specified, and the header and index are True, then the index names are used. If DataFrame uses MultiIndex, a sequence should be given.
* **startrow:** Default value 0. It refers to the upper left cell row to dump DataFrame.
* **startcol:** Default value 0. It refers to upper left cell column to dump DataFrame.
* **engine:** It is optional parameter that writes engine to use, openpyxl, or xlsxwriter.
* **merge\_cells:** It returns the boolean value and its default value is True. It writes MultiIndex and Hierarchical rows as the merged cells.
* **encoding:** It is an optional parameter that encodes the resulting excel file. It is only necessary for the xlwt.
* **inf\_rep:** It is also an optional parameter and its default value is inf. It usually represents infinity.
* **verbose:** It returns a boolean value. It's default value is True.It is used to display more information in the error logs.
* **freeze\_panes:** It is also an optional parameter that specifies the one based bottommost row and rightmost column that is to be frozen.

1. **import** pandas as pd
2. info\_marks = pd.DataFrame({'name': ['Parker', 'Smith', 'William', 'Terry'], 'Maths': [78, 84, 67, 72], 'Science': [89, 92, 61, 77], 'English': [72, 75, 64, 82]})
3. writer = pd.ExcelWriter('output.xlsx') # render dataframe as html
4. info\_marks.to\_excel(writer)
5. writer.save()

# Pandas DataFrame.transform

We can define Pandas DataFrame as a two-dimensional size-mutable, heterogeneous tabular data structure with some labeled axes (rows and columns). Performing the arithmetic operations will align both row and column labels. It can be considered as a dict-like container for Series objects.

The main task of Pandas **DataFrame.transform()** function is to self produce a DataFrame with its transformed values and it has the same axis length as self.

### Syntax: DataFrame.transform(func, axis=0, \*args, \*\*kwargs)

**func :** It is a function that is used for transforming the data.

**axis :** Refers to 0 or 'index', 1 or 'columns', default value 0.

**\*args:** It is a positional arguments that is to be passed to a func.

**\*\*kwargs :** It is a keyword arguments that is to be passed to a func.

### Returns:It returns the DataFrame that must have same length as self.

**Exa 1 :** Use DataFrame.transform() function to add 10 to each element in the dataframe.

1. info =pd.DataFrame({"P":[8, 2, 9, None, 3], "Q":[4, 14, 12, 22, None],
2. "R":[2, 5, 7, 16, 13], "S":[16, 10, None, 19, 18]})
3. index\_ =['A\_Row', 'B\_Row', 'C\_Row', 'D\_Row', 'E\_Row'] # Create the index
4. info.index =index\_ ,/ print(info)# Set the index

# Pandas DataFrame.transpose()

The transpose() function helps to transpose the index and columns of the dataframe. It reflects DataFrame over its main diagonal by writing rows as columns and vice-versa.

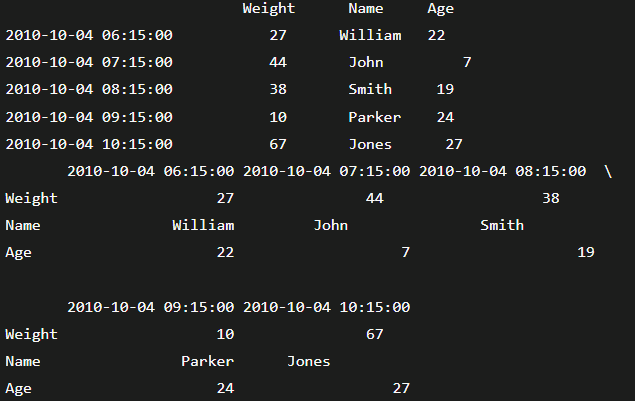
### Syntax: dataFrame.transpose(\*args, \*\*kwargs)

**copy:** If its value is True, then the underlying data is being copied. Otherwise, by default, no copy is made, if possible.

**\*args, \*\*kwargs:** Both are additional keywords that do not affect, but has an acceptance that provide compatibility with a numpy.

### Returns: It returns the transposed DataFrame.

1. info = pd.DataFrame({'Weight':[27, 44, 38, 10, 67], 'Name':['William', 'John', 'Smith', 'Parker', 'Jones'], 'Age':[22, 17, 19, 24, 27]})
2. index\_ = pd.date\_range('2010-10-04 06:15', periods = 5, freq ='H') # Create index
3. info.index = index\_ # Set the index
4. print(info)
5. result = info.transpose() # **return** the transpose



# Pandas DataFrame.where()

The main task of the **where()** method is to check the data frame for one or more conditions and return the result accordingly. By default, if the rows are not satisfying the condition, it is filled with **NaN** value.

**DataFrame.where(cond, other=nan, inplace=False, axis=None, level=None, errors='raise', try\_cast=False, raise\_on\_error=None)**

* **cond:** It refers to one or more conditions to check the data frame.
* **other:** It replaces the rows that do not satisfy the condition with the user-defined object; the default value is NaN.
* **inplace:** Returns the boolean value. If the value is true, it makes the changes in the dataframe itself.
* **axis:** An axis to check( row or columns)

1. **import** pandas as pd
2. **import** numpy as np
3. a = pd.Series(range(5))
4. a.where(a > 0)
5. a.mask(a > 0)
6. a.where(a > 1, 10)
7. info = pd.DataFrame(np.arange(10).reshape(-1, 2), columns=['A', 'B'])
8. info
9. b = info % 3 == 0
10. info.where(b, -info)
11. info.where(b, -info) == np.where(b, info, -info)
12. info.where(b, -info) == info.mask(~b, -info)

# Add a column to DataFrame Columns

We can add a new column to an existing DataFrame using different ways. For the demonstration, first, we have to write a code to read the existing file, which consists of some columns in a DataFrame.

1. **import** pandas as pd
2. aa = pd.read\_csv("aa.csv")
3. aa.head()

## Add new columns to a DataFrame using [] operator

If we want to add any new column at the end of the table, we have to use the **[]** operator. Let's add a new column named "**Age**" into "**aa**" csv file.

1. **import** pandas as pd
2. aa = pd.read\_csv("aa.csv")
3. aa["Age"] = "24"
4. aa.head()

**For ex:** aa["Designation"]

In above code, Pandas will throw an error because **Designation** column does not exist.

But if we assign a value to that column, Pandas will generate a new column automatically at the end of the table.

## Add new columns in a DataFrame using insert()

We can also add a new column at any position in an existing DataFrame using a method name **insert**.

1. **import** pandas as pd
2. aa = pd.read\_csv("aa.csv")
3. aa.insert(2, column = "Department", value = "B.Sc")
4. aa.head()



# Convert Pandas DataFrame to Numpy array

For performing some high-level mathematical functions, we can convert Pandas DataFrame to numpy arrays. It uses the DataFrame.to\_numpy() function.

**DataFrame.to\_numpy()** function is applied on DataFrame that returns numpy ndarray.

### Syntax: DataFrame.to\_numpy(dtype=None, copy=False)

* **dtype:** It is an optional parameter that pass the dtype to numpy.asarray().
* **copy:** It returns the boolean value that has the default value **False**.

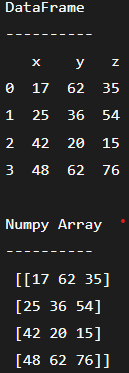
It ensures that the returned value is not a view on another array.

### Returns: It returns the **numpy.ndarray** as an output.

1. info = pd.DataFrame({"P": [2, 3], "Q": [4.0, 5.8]})
2. info.to\_numpy()
3. info['R'] = pd.date\_range('2000', periods=2)
4. info.to\_numpy()



1. info = pd.DataFrame([[17, 62, 35],[25, 36, 54],[42, 20, 15],[48, 62, 76]],
2. columns=['x', 'y', 'z'])
3. print('DataFrame\n----------\n', info)
4. arr = info.to\_numpy()
5. print('\nNumpy Array\n----------\n', arr)



# Convert Pandas DataFrame to CSV

The Pandas **to\_csv()** function is used to convert the DataFrame into CSV data. To write the CSV data into a file, we can simply pass a file object to the function. Otherwise, the CSV data is returned in a string format.

**DataFrame.to\_csv(path\_or\_buf=None, sep=', ', na\_rep='', float\_format=None, columns=None, header=True, index=True, index\_label=None, mode='w', encoding=None, compression='infer', quoting=None, quotechar='"', line\_terminator=None, chunksize=None, date\_format=None, doublequote=True, escapechar=None, decimal='.')**

## Parameters:

## **path\_or\_buf:** It refers to **str** or **file handle**. Basically, it defines the path of file or object. The default value is **None** and if **None** value is passed, then it returns a string value.

If we pass file object, it should be opened with **newline ="** & disable universal newlines.

**sep:** It refers to **string** value & consists a string of length 1. It's default valu is **comma**(**,**).

**na\_rep:** It refers to a string value that represents null or missing values. The **empty string** is the default value.

**float\_format:** It also consists a **string** value that is responsible for formatting a string for the floating-point numbers.

**columns:** It is an optional parameter that refers to a sequence to specify the columns that need to be included in the CSV output.

**header:** It generally consists a boolean value or a list of string. If its value is set to False, then the column names are not written in the output. It's default value is True.

If we pass a list of string as input, it generally writes column names in the output. length of the list of the file should be same as number of columns being written in the CSV file.

**index:** If the value is set to True, index is included in the CSV data. Otherwise, the index value is not written in CSV output.

**index\_label:** It consists a **str** value or a sequence that is used to specify the column name for index. It's default value is None.

**mode:** It refers a **string** value that is used for writing mode. It's default value is **w.**

**encoding:** It is an **optional** parameter consisting a string value that represents an encoding used in the output file. The default value of encoding is **UTF-8.**

**compression:** It refers a **str** value that compress the mode among the following values{'infer', 'gzip', 'bz2', 'zip', 'xz', None}. It detects the compression from the extensions: **'.gz', '.bz2', '.zip' or '.xz'** if **infer** and **path\_or\_buf** is a path-like, otherwise no compression occurs.

**quoting:** It is an **optional** parameter that is defined as a constant from the csv module. Its default value is **csv.QUOTE\_MINIMAL**. If you set a float\_format then floating value is converted to strings and **csv.QUOTE\_NONNUMERIC** is treated as a non-numeric value.

**quotechar:** It refers to a **str** value of length 1. It is character that is used to quote fields.

**line\_terminator:** It is an **optional** parameter that refers to a **string** value. Its main task is to terminate the line. It is a newline character that is to be used in the output file. Its default value is set to **os.linesep** that mainly depends on the OS. An individual method is called to define the operating system ('**n**' for **linux**, '**rn'** for '**Windows'**).

A**chunksize:** It consists **None** or **integer** value & define rows to write at the current time.

**date\_format:** It consists **str** value and used to format a string for the datetime objects. The default value of **date\_format** is **None**.

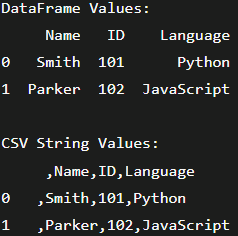
**doublequote:** It consists a boolean value that have the default value **True**. It is mainly used for controlling the quote of **quotechar** inside the field.

**escapechar:** It consists a **string** value of length 1. Basically, it is a character that is used to escape **sep** and **quotechar**. The default value of **escapechar** is **None**.

**decimal:** It consists a **string** value that identify a character as decimal separator. **Ex**: use '**,**' for European data.

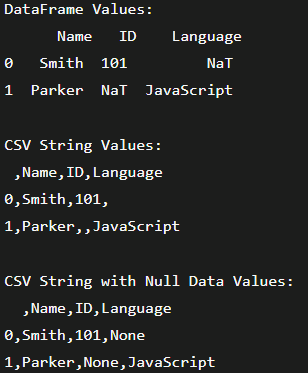
## Returns: It returns **str** or **Non**e value. If a parameter value named as **path\_or\_buf** is None, it returns the resulting csv format as a string. Otherwise, it returns None.

1. **import** pandas as pd
2. data = {'Name': ['Smith', 'Parker'], 'ID': [101, 102], 'Language': ['Python', 'JavaScript']}
3. info = pd.DataFrame(data)
4. print('DataFrame Values:\n', info)
5. csv\_data = info.to\_csv() # **default** CSV
6. print('\nCSV String Values:\n', csv\_data)



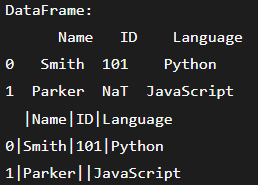
**Ex2:** below example shows Null or missing Data Representation in the CSV Output file:

1. **import** pandas as pd
2. data = {'Name': ['Smith', 'Parker'], 'ID': [101, pd.NaT], 'Language': [pd.NaT, 'JavaScript']}
3. info = pd.DataFrame(data)
4. print('DataFrame Values:\n', info)
5. csv\_data = info.to\_csv()
6. print('\nCSV String Values:\n', csv\_data)
7. csv\_data = info.to\_csv(na\_rep="None")
8. print('CSV String with Null Data Values:\n', csv\_data)



**Example3:** The below example specify the delimiter for the CSV output.

1. data = {'Name': ['Smith', 'Parker'], 'ID': [101, pd.NaT], 'Language': [Python, 'JavaScript']}
2. info = pd.DataFrame(data)
3. print('DataFrame:\n', info)
4. csv\_data = info.to\_csv(sep='|')
5. print(csv\_data)



# Python Pandas Reading Files

## Reading from CSV File

For reading the Pandas files, firstly we have to load data from file formats into a DataFrame. You need only a single line to load your data in code.

1. **import** pandas
2. df = pandas.read\_csv('hrdata.csv')
3. **print**(df)

## Reading from JSON

If you have any JSON file, Pandas can easily read it through a single line of code.

1. df =pd.read\_json('hrdata.json')

Pandas convert a list of lists into a DataFrame and also define the column names separately. A JSON parser is responsible for converting a JSON text into another representation that must accept all the texts according to the JSON grammar. It can also accept non JSON forms or extensions.

## 

## Reading from the SQL database

For reading a file from SQL, first, you need to establish a connection using the Python library and then pass the query to pandas. Here, we use SQLite for demonstration.

**sqlite3** is used to establish a connection to the database, and then we can use it to generate a DataFrame through **SELECT** query.

For establishing a connection to the SQLite database file:

1. **import** sqlite3
2. con = sqlite3.connect("database.db")

A table called **information** is present in the SQLite database, and the index of the column called "index". We can read data from the **information** table by passing the **SELECT** query and the **con**.

df = pd.read\_sql\_query("SELECT \* FROM information", con)

# Pandas Concatenation

Pandas is capable of combining Series, DataFrame, and Panel objects through different kinds of set logic for the indexes and the relational algebra functionality.

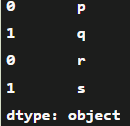
The **concat()** function is responsible for performing concatenation operation along an axis in the DataFrame.

**Syntex:-** pd.concat(objs,axis=0,join='outer',join\_axes=None, ignore\_index=False)

* **objs:** It is a sequence or mapping of series or DataFrame objects. If we pass a dict in the DataFrame, then the sorted keys will be used as the keys<.strong> argument, and the values will be selected in that case. If any non-objects are present, then it will be dropped unless they are all none, and in this case, a ValueError will be raised.
* **axis:** It is an axis to concatenate along**.**
* **join:** Responsible for handling indexes on another axis.
* **join\_axes:** A list of index objects. Instead of performing the inner or outer set logic, specific indexes use for the other (n-1) axis.
* **ignore\_index:** bool, default value False.It does not use the index values on the concatenation axis, if true. The resulting axis will be labeled as 0, ..., n - 1.

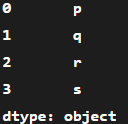
## Returns: A series is returned when we concatenate all the Series along the axis (axis=0). In case if objs contains at least one DataFrame, it returns a DataFrame.

1. **import pandas as pd**
2. **a\_data = pd.Series(['p', 'q'])**
3. **b\_data = pd.Series(['r', 's'])**
4. **pd.concat([a\_data, b\_data])**



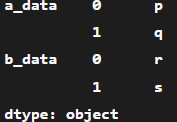
**Ex2:** In the above ex-, we can reset the existing index by using the ignore\_index parameter. The below code demonstrates the working of ignore\_index.

1. **import pandas as pd**
2. **a\_data = pd.Series(['p', 'q'])**
3. **b\_data = pd.Series(['r', 's'])**
4. **pd.concat([a\_data, b\_data], ignore\_index=True)**



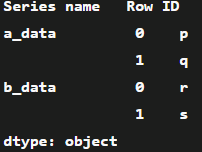
**Ex 3:** We can add a hierarchical index at the outermost level of the data by using the keys parameter.

1. **import pandas as pd**
2. **a\_data = pd.Series(['p', 'q'])**
3. **b\_data = pd.Series(['r', 's'])**
4. **pd.concat([a\_data, b\_data], keys=['a\_data', 'b\_data'])**



1. **import pandas as pd**
2. **a\_data = pd.Series(['p', 'q'])**
3. **b\_data = pd.Series(['r', 's'])**
4. **pd.concat([a\_data, b\_data], keys=['a\_data', 'b\_data'])**
5. **pd.concat([a\_data, b\_data], keys=['a\_data', 'b\_data'],**

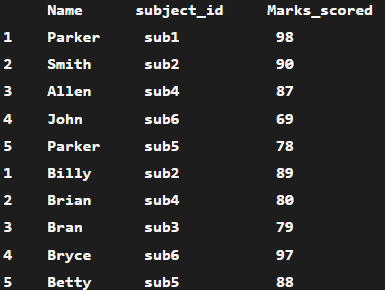
**names=['Series name', 'Row ID'])**



### Concatenation using append

**Append method is defined as a useful shortcut to concatenate Series and DataFrame.**

1. **import pandas as pd**
2. **one = pd.DataFrame({ 'Name': ['Parker', 'Smith', 'Allen', 'John', 'Parker'], 'subject\_id':['sub1','sub2','sub4','sub6','sub5'], 'Marks\_scored':[98,90,87,69,78]},**
3. **index=[1,2,3,4,5])**
4. **two = pd.DataFrame({'Name': ['Billy', 'Brian', 'Bran', 'Bryce', 'Betty'],**
5. **'subject\_id':['sub2','sub4','sub3','sub6','sub5'], 'Marks\_scored':[89,80,79,97,88]},**
6. **index=[1,2,3,4,5])**
7. **print (one.append(two))**



# Python Pandas Data operations

In Pandas,there are different useful data oprations for DataFrame, which are as follows :

**Row and column selection:** We can select any row and column of the DataFrame by passing the name of the rows and column. When you select it from the DataFrame,

it becomes one-dimensional and considered as Series.

**Filter Data:** We can filter data by providing some of boolean expression in DataFrame.

#### Note: If we want to pass the boolean results into a DataFrame, then it shows all the results.

**Null values:** A Null value can occur when no data is being provided to the items. The various columns may contain no values which are usually represented as NaN. In Pandas, several useful functions are available for detecting, removing, and replacing the null values in Data Frame. These functions are as follows:

**isnull():** The main task of isnull() is to return the true value if any row has null values.

**notnull():** It is opposite of isnull() function and it returns true values for not null value.

**dropna():** This method analyzes and drops the rows/columns of null values.

**fillna():** It allows the user to replace the NaN values with some other values.

**replace():** It is a very rich function that replaces a string, regex, series, dictionary, etc.

**interpolate():** It is a very powerful function that fills null values in DataFrame or series.

**String operation:**A set of a string function is available in Pandas to operate on string data and ignore the missing/NaN values. There are different string operation that can be performed using **.str.** option. These functions are as follows:

**lower():** It converts any strings of the series or index into lowercase letters.

**upper():** It converts any string of the series or index into uppercase letters.

**strip():** This function helps to strip the whitespaces including a new line from each string in the Series/index.

**split(' '):** It is a function that splits the string with the given pattern.

**cat(sep=' '):** It concatenates series/index elements with a given separator.

**contains(pattern):** It returns True if a substring is present in the element, else False.

**replace(a,b):** It replaces the value a with the value b.

**repeat(value):** It repeats each element with a specified number of times.

**count(pattern):** It returns the count of the appearance of a pattern in each element.

**startswith(pattern):** It returns True if all the elements in the series starts with a pattern.

**endswith(pattern):** It returns True if all the elements in the series ends with a pattern.

**find(pattern):** It is used to return the first occurrence of the pattern.

**findall(pattern):** It returns a list of all the occurrence of the pattern.

**swapcase:** It is used to swap the case lower/upper.

**islower():** It returns True if all the characters in the string of the Series/Index are in lowercase. Otherwise, it returns False.

**isupper():** It returns True if all the characters in the string of the Series/Index are in uppercase. Otherwise, it returns False.

**isnumeric():** It returns True if all the characters in the string of the Series/Index are numeric. Otherwise, it returns False.

**Count Values:** This operation is used to count the total number of occurrences using '**value\_counts()**' option.

**Plots:** Pandas plots the graph with the **matplotlib** library. The **.plot()** method allows you to plot the graph of your data.

**.plot()** function plots index against every column. You can also pass the arguments into the **plot()** function to draw a specific column.

# Data processing

Most of the time of data analysis and modeling is spent on data preparation and processing i.e., loading, cleaning and rearranging the data, etc. Further, because of Python libraries, Pandas give us high performance, flexible, and high-level environment for processing the data. Various functionalities are available for pandas to process the data effectively.

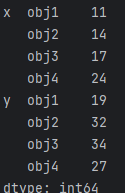
**Hierarchical indexing**

For enhancing the capabilities of Data Processing, we have to use some indexing that helps to sort the data based on the labels. So, Hierarchical indexing is comes into the picture and defined as an essential feature of pandas that helps us to use the multiple index levels.

**Creating multiple index**

In Hierarchical indexing, we have to create multiple indexes for the data. This example creates a series with multiple indexes.

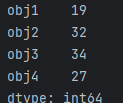
1. info = pd.Series([11, 14, 17, 24, 19, 32, 34, 27],
2. index = [['x', 'x', 'x', 'x', 'y', 'y', 'y', 'y'],
3. ['obj1', 'obj2', 'obj3', 'obj4', 'obj1', 'obj2', 'obj3', 'obj4']])
4. print(info)



Partial indexing

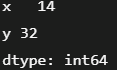
Partial indexing can be defined as a way to choose the particular index from a hierarchical indexing. Below code is extracting 'b' from the data,

1. info = pd.Series([11, 14, 17, 24, 19, 32, 34, 27],
2. index = [['x', 'x', 'x', 'x', 'y', 'y', 'y', 'y'],
3. ['obj1', 'obj2', 'obj3', 'obj4', 'obj1', 'obj2', 'obj3', 'obj4']])
4. print(info['y'])



Further, the data can also be extracted based on inner level i.e. 'obj'. The below result defines two available values for 'obj2' in the Series.

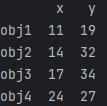
**info[**:, 'obj2']



### Unstack the data

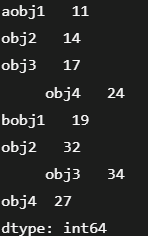
Unstack means to change the row header to the column header. The row index will change to the column index, therefore the Series will become the DataFrame.

1. info = pd.Series([11, 14, 17, 24, 19, 32, 34, 27], index = [['x', 'x', 'x', 'x', 'y', 'y', 'y', 'y'],
2. ['obj1', 'obj2', 'obj3', 'obj4', 'obj1', 'obj2', 'obj3', 'obj4']])
3. # unstack on first level i.e. x, y #note that data row-labels are x and y
4. info.unstack(0)

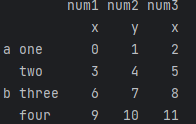


'**stack()**' operation is used to convert the column index to row index. In above code, we can convert 'obj' as column index into row index using '**stack**' operation.

1. info = pd.Series([11, 14, 17, 24, 19, 32, 34, 27], index = [['x', 'x', 'x', 'x', 'y', 'y', 'y', 'y'],
2. ['obj1', 'obj2', 'obj3', 'obj4', 'obj1', 'obj2', 'obj3', 'obj4']])
3. # unstack on first level i.e. x, y #note that data row-labels are x and y
4. data.unstack(0)
5. d.stack()



1. **import** numpy as np
2. info = pd.DataFrame(np.arange(12).reshape(4, 3),
3. index = [['a', 'a', 'b', 'b'], ['one', 'two', 'three', 'four']],
4. columns = [['num1', 'num2', 'num3'], ['x', 'y', 'x']] ... )
5. info



### Swap and sort level

We can easily swap the index level by using '**swaplevel**' command, which takes input as two level-numbers.

# Pandas DataFrame.corr()

The main task of the **DataFrame.corr()** method is to find the pairwise correlation of all the columns in DataFrame. If any null value is present, it will automatically be excluded.

It also ignores non-numeric data type columns from the DataFrame.

Syntax: DataFrame.count(axis=0, level=None, numeric\_only=False)

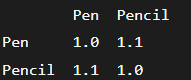
### Parameters **method:**

* **pearson:** standard correlation coefficient.
* **kendall:** Kendall Tau correlation coefficient.
* **spearman:** Spearman rank correlation.
* **callable:** callable with input two 1d ndarrays that returns a float value.

**min\_periods:** It is an optional parameter that requires a minimum number of observations per pair of columns to return a valid result. Currently it is only available for the **Pearson and Spearman correlation**.

### Returns: It returns a DataFrame correlation matrix.

1. Def histogram\_intersection(x, y):
2. a = np.minimum(x, y).sum().round(decimals=1)
3. **return** a
4. info = pd.DataFrame([(.6, .2), (.4, .7), (.3, .5), (.5, .2)], columns=['Pen', 'Pencil'])
5. info.corr(method=histogram\_intersection)



# Pandas DataFrame.dropna()

If your dataset consists of null values, we can use the dropna() function to analyze and drop the rows/columns in the dataset.

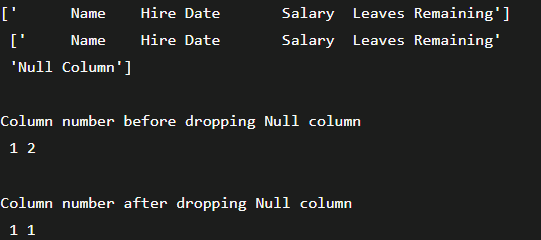
## Syntax: DataFrameName.dropna(axis=0, how='any', thresh=None, subset=None, inplace=False)

* **axis :** {0 or 'index', 1 or 'columns'}, default value 0. It takes **int** or **string** values for rows/columns. input can be 0 & 1 for integers & **index** or **columns** for the string.
  + **0, or 'index':** Drop the rows which contain missing values.
  + **1, or 'columns':** Drop the columns which contain the missing value.
* **how :**It determines if row or column is removed from DataFrame when we have at least one NA or all NA. It takes a string value of only two kinds ('any' or 'all').
  + **any:** It drops the row/column if any value is null.
  + **all:** It drops only if all values are null.
* **thresh:**It takes integer value that defines minimum amount of NA values to drop.
* **Subset:** It is an array that limits the dropping process to passed rows/columns through the list.
* **Inplace:** It returns a boolean value that makes changes in DF itself if its True.

## Returns: It returns the DataFrame from which NA entries has been dropped.

For Demonstration, first, we are taking a csv file that will drop any column from dataset.

1. info = pd.read\_csv("aa.csv") # making data frame from csv file
2. copy = pd.read\_csv("aa.csv") # making a copy of old data frame
3. copy["Null Column"]= None #creating value with all **null** values in **new** data frame
4. # checking **if** column is inserted properly
5. print(info.columns.values, "\n", copy.columns.values)
6. print("\nColumn number before dropping Null column\n", len(info.dtypes), len(copy.dtypes)) # comparing values before dropping **null** column
7. copy.dropna(axis = 1, how ='all', inplace = True) #droping colmn with all **null** vals
8. print("\nColumn number after dropping Null column\n", len(info.dtypes), len(info.dtypes)) # comparing values after dropping **null** column



The above code dropped the null column from the dataset and returned a new DataFrame.

# Pandas DataFrame.fillna()

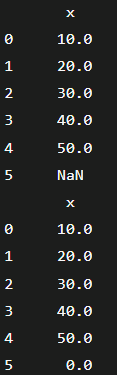
We can use the fillna() function to fill the null values in the dataset.

## Syntax:DataFrame.fillna(value=None, method=None, axis=None, inplace=False, limit=None, downcast=None, \*\*kwargs)

* **value:** It’s a valu that is used to fill null values, alternately Series/dict/DataFrame.
* **method:** A method that is used to fill the null values in the reindexed Series.
* **axis:** It takes int or string value for rows/columns. Axis along which we need to fill missing values.
* **inplace:** If it is True, it fills values at an empty place.
* **limit:** It is an integer value that specifies the maximum number of consecutive forward/backward NaN value fills.
* **downcast:** It takes a dict that specifies what to downcast like Float64 to int64.

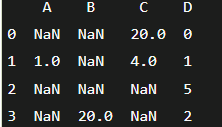
## Returns:It returns an object in which the missing values are being filled.

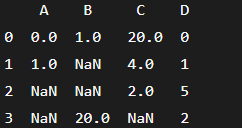
1. info = pd.DataFrame(data={'x':[10,20,30,40,50,None]})
2. print(info)
3. info.fillna(value=0, inplace=True) # Fill **null** value to dataframe using 'inplace'
4. print(info)



### Ex:- Below code is responsible for filling the DataFrame that consist some NaN values.

1. info = pd.DataFrame([[np.nan,np.nan, 20, 0], [1, np.nan, 4, 1], [np.nan, np.nan, np.nan, 5], [np.nan, 20, np.nan, 2]], columns=list('ABCD'))
2. info
3. info.fillna(0)
4. info.fillna(method='ffill')
5. values = {'A': 0, 'B': 1, 'C': 2, 'D': 3}
6. info.fillna(value=values)
7. info.fillna(value=values, limit=1)





# Pandas DataFrame.replace()

Pandas replace() is a very rich function that is used to replace a **string, regex, dictionary, list,** and **series** from the DataFrame. The values of the DataFrame can be replaced with other values dynamically. It is capable of working with the Python regex.

It differs from updating with **.loc** or **.iloc**, which requires you to specify a location where you want to update with some value.

## Syntax: DataFrame.replace(to\_replace=None, value=None, inplace=False, limit=None, regex=False, method='pad', axis=None)

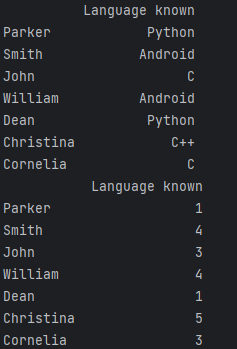
* **to\_replace:** Defines a pattern that we are trying to replace in dataframe.
* **value:** It is a value that is used to fill holes in the DataFrame (e.g., 0), alternately a dict of values that specify which value to use for each column (columns not in the dict will not be filled).  
  It also allow such objects of regular expressions, strings, and lists or dicts, etc.
* **inplace:** If it is True, then it replaces in place.

#### Note: It will also modify any other views on this object (e.g., a column from a DataFrame). Returns the caller if this is True.

* **limit:** It defines the maximum size gap to forward or backward fill.
* **regex:** It check whether to interpret to\_replace and/or val as regular expressions. If it is True, then to\_replace must be a string. Otherwise, **to\_replace** must be None because this parameter will be interpreted as a regular expression or a list, dict, or array of regular expressions.
* **method:** It is a method to use for replacement when to\_replace is a list.

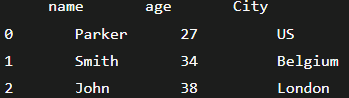
**Returns:** It returns a DataFrame object after the replacement.

1. info = pd.DataFrame({'Language known': ['Python', 'Android', 'C', 'Android', 'Python', 'C++', 'C']},
2. index=['Parker', 'Smith', 'John', 'William', 'Dean', 'Christina', 'Cornelia'])
3. print(info)
4. dictionary = {"Python": 1, "Android": 2, "C": 3, "Android": 4, "C++": 5}
5. info1 = info.replace({"Language known": dictionary})
6. print(info1)



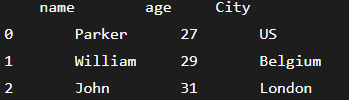
### Ex2:The below example replaces a value with another in a DataFrame.

1. info = pd.DataFrame({ 'name':['Parker','Smith','John'],
2. 'age':[27,34,29], 'city':['US','Belgium','London'] })
3. info.replace([29],38)



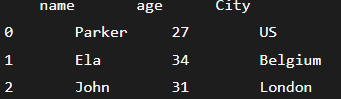
### Ex3: The below example replaces the values from a dict:

1. info = pd.DataFrame({ 'name':['Parker','Smith','John'],
2. 'age':[27,34,31], 'city':['US','Belgium','London'] })
3. info.replace({ 34:29, 'Smith':'William' })



### Ex4:The below example replaces the values from regex:

1. info = pd.DataFrame({ 'name':['Parker','Smith','John'],
2. 'age':[27,34,31], 'city':['US','Belgium','London'] })
3. info.replace('Sm.+','Ela',regex=True)



# Pandas DataFrame.iloc[]

The **DataFrame.iloc[]** is used when the index label of the DataFrame is other than numeric series of 0,1,2,....,n or in the case when the user does not know the index label.

We can extract the rows by using an imaginary index position which is not visible in the DataFrame. It is an integer- based position(from 0 to length-1 of the axis), but may also be used with the boolean array.

The allowed inputs for **.loc[]** are:

* Integer value, e.g. 7.
* List or array of integers, e.g [2, 5, 6].
* Slice object with ints, e.g., 1:9.
* boolean array.
* A callable function with one argument that can be calling Series or DataFrame. It returns valid outputs for indexing.

It can raise the **IndexError** if we request the index is out-of-bounds, except slice indexers, which allow the out-of-bounds indexing.

### Syntax: pandas.DataFrame.iloc[]

# Pandas DataFrame.isin()

The main task of the **DataFrame.isin()** method is to select the rows having a particular (or multiple) values in a particular column.

### Syntax: DataFrame.isin(values)

### Parameter:

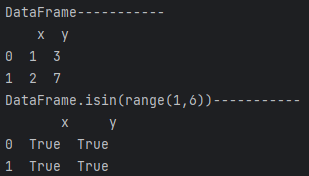
### **values:** It can be DataFrame, Series, Iterable, or dict and returns a boolean value.

It returns a true value if all labels match. If it consists of a **Series**, then it will be index.

If it consists of a **dict**, then the keys must be the column names and must be matched.

If it consists of a DataFrame, then both the index and column labels must be matched.

1. info = pd.DataFrame({'x': [1, 2], 'y': [3, 7]})
2. p = info.isin(range(1,8)) #check **if** the values of info are in the range(1,6)
3. print('DataFrame\n-----------\n',info)
4. print('\nDataFrame.isin(range(1,6))\n-----------\n',p)



1. data = pd.DataFrame({ 'EmpCode': ['Emp001', 'Emp002', 'Emp003', 'Emp004', 'Emp005'], 'Name': ['Parker', 'Smith', 'Jones', 'Terry', 'Palin'], 'Occupation': ['Tester', 'Developer', 'Statistician', 'Tester', 'Developer'], 'Date Of Join': ['2019-01-17', '2019-01-26', '2019-01-29', '2019-02-02', '2019-02-11'], 'Age': [29, 22, 25, 38, 27]})
2. print(data.loc[data['Occupation'].isin(['Tester','Developer'])])
3. print("\nMultiple Conditions\n")
4. print(data.loc[(data['Occupation'] == 'Tester') | (data['Name'] == 'John') & (data['Age'] < 27)])



# Pandas DataFrame.loc[]

The **DataFrame.loc[]** is used to retrieve the group of rows and columns by labels or a boolean array in the DataFrame. It takes only index labels, and if it exists in the caller DataFrame, it returns the rows, columns, or DataFrame.

The **DataFrame.loc[]** is a label based but may use with the boolean array.

The allowed inputs for **.loc[]** are:

* Single label, e.g.,**7** or **a**. Here, **7** is interpreted as the label of the index.
* List or array of labels, e.g. ['x', 'y', 'z'].
* Slice object with labels, e.g. 'x':'f'.
* A boolean array of the same length. e.g. [True, True, False].
* **callable** function with one argument.

### Syntax: pandas.DataFrame.loc[]

### Returns: It returns Scalar, Series or DataFrame.

1. info = pd.DataFrame({'Age':[32, 41, 44, 38, 33], 'Name':['Phill', 'William', 'Terry', 'Smith', 'Parker']})
2. index\_ = ['Row\_1', 'Row\_2', 'Row\_3', 'Row\_4', 'Row\_5'] # Create the index
3. info.index = index\_ # Set the index
4. **final** = info.loc['Row\_2', 'Name'] # **return** the value
5. print(**final**) //William
6. info = pd.DataFrame({"P":[28, 17, 14, 42, None],
7. "Q":[15, 23, None, 15, 12],
8. "R":[11, 23, 16, 32, 42],
9. "S":[41, None, 34, 25, 18]})
10. # Create the index
11. index\_ = ['A', 'B', 'C', 'D', 'E']
12. # Set the index
13. info.index = index\_
14. # Print the DataFrame
15. print(info)

### Ex: info = pd.DataFrame({"P":[28, 17, 14, 42, None], "Q":[15, 23, None, 15, 12],

### "R":[11, 23, 16, 32, 42], "S":[41, None, 34, 25, 18]})

### index\_ = ['A', 'B', 'C', 'D', 'E'] # Create the index

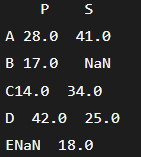
### info.index = index\_ # Set the index

### print(info)

### 

Now, we have to use **DataFrame.loc** attribute to return the values present in the DataFrame.

1. result = info.loc[:, ['P', 'S']] # **return** the values
2. print(result)



# Pandas loc vs. iloc

Pandas offers .loc[] and .iloc[] methods for **data slicing**. Data Slicing generally refers to inspect your data sets. These two methods belong to the index selection method that is used to set an identifier for each row of the data set. The indexing can take specific labels, and these labels can either be an integer or any other value specified by the user.

The .**loc[]** method is used to retrieve the group of rows and columns by labels or a boolean array present in the DataFrame. It takes only index labels, and if it exists in the caller DataFrame, it returns the rows, columns, or DataFrame. It is a label-based method but may be used with the boolean array.

Whereas, the **.iloc[]** method is used when the index label of the DataFrame is other than numeric series of 0,1,2,....,n, or in the case when the user does not know the index label.

There are some differences between the above methods, which are given below:

1. **.loc[]** method is a **label based** method that means it takes names or labels of the index when taking the slices, whereas **.iloc[]** method is based on the **index's position**. It behaves like a regular slicing where we just have to indicate positional index number and simply get the appropriate slice.
2. The .**loc[]** method includes the last element of the table whereas **.iloc[]** method does not include the last element.
3. The .**loc[]** method is a **name-based indexing,** whereas the **.iloc[]** method is **positional based indexing**.
4. The arguments of .**iloc[]** can be:
   * list of rows and columns
   * range of rows and columns
   * single row and column
5. Whereas, the arguments of **.loc[]** can be:
   * row label
   * list of row label
6. The **.loc[]** method indexer can perform the boolean selection by passing the boolean series, but in the case of **.iloc[]**method, we cannot pass a boolean series.

#############################################################

# Pandas Cheat Sheet

Pandas can be used as the most important Python package for Data Science. It helps to provide a lot of functions that deal with the data in easier way.

## Key and Imports: We use following shorthand in the cheat sheet:

* **df:** Refers to any Pandas Dataframe object.
* **s:** Refers to any Pandas Series object. You can use below imports to get started:

### Importing Data

* **pd.read\_csv(filename) :** It read the data from CSV file.
* **pd.read\_table(filename) :** It is used to read the data from delimited text file.
* **pd.read\_excel(filename) :** It read the data from an Excel file.
* **pd.read\_sql(query,connection \_object) :** It read data from a SQL table/database.
* **pd.read\_json(json \_string) :** It read data from a JSON formatted string,URL or file.
* **pd.read\_html(url) :** It parses an html URL, string or the file and extract the tables to a list of dataframes.
* **pd.read\_clipboard() :** It takes the contents of clipboard and passes it to the read\_table() function.
* **pd.DataFrame(dict) :** From the dict, keys for the columns names, values for the data as lists.

### Exporting data

* **df.to\_csv(filename):** It writes to a CSV file.
* **df.to\_excel(filename):** It writes to an Excel file.
* **df.to\_sql(table\_name, connection\_object):** It writes to a SQL table.
* **df.to\_json(filename) :** It write to a file in JSON format.

### Create Test objects: It is useful for testing the code segments.

* **pd.DataFrame(np.random.rand(7,18)):** Refers to 18 columns and 7 rows of random floats.
* **pd.Series(my\_list):** It creates a Series from an iterable my\_list.
* **df.index= pd.date\_range('1940/1/20', periods=df.shape[0]):** It adds date index.

### Viewing/Inspecting Data

* **df.head(n):** It returns first n rows of the DataFrame.
* **df.tail(n):** It returns last n rows of the DataFrame.
* **df.shape:** It returns number of rows and columns.
* **df.info():** It returns index, Datatype, and memory information.
* **s.value\_counts(dropna=False):** It views unique values and counts.
* **df.apply(pd.Series.value\_counts):** It refers to unique vals & counts for all colmns.

### Selection

* **df[col1]:** It returns column with the label col as Series.
* **df[[col1, col2]]:** It returns columns as a new DataFrame.
* **s.iloc[0]:** It select by the position.
* **s.loc['index\_one']:** It select by the index.
* **df.iloc[0,:]:** It returns first row.
* **df.iloc[0,0]:** It returns the first element of first column.

### Data cleaning

* **df.columns = ['a','b','c']:** It rename the columns.
* **pd.isnull():** It checks for the null values and returns the Boolean array.
* **pd.notnull():** It is opposite of pd.isnull().
* **df.dropna():** It drops all the rows that contain the null values.
* **df.dropna(axis= 1):** It drops all the columns that contain null values.
* **df.dropna(axis=1,thresh=n):** It drops all rows that have less than n non null valus.
* **df.fillna(x):** It replaces all null values with x.
* **s.fillna(s.mean()):** It replaces all the null values with the mean(the mean can be replaced with almost any function from the statistics module).
* **s.astype(float):** It converts the datatype of series to float.
* **s.replace(1, 'one'):** It replaces all the values equal to 1 with 'one'.
* **s.replace([1,3],[ 'one', 'three']):**It replaces all 1 with 'one' and 3 with 'three'.
* **df.rename(columns=lambda x: x+1):**It rename mass of the columns.
* **df.rename(columns={'old\_name': 'new\_ name'}):** It consist selective renaming.
* **df.set\_index('column\_one'):** Used for changing the index.
* **df.rename(index=lambda x: x+1):** It rename mass of the index.

### Filter, Sort, and Groupby

* **df[df[col] > 0.5]:** Returns the rows where column col is greater than 0.5
* **df[(df[col] > 0.5) & (df[col] < 0.7)] :** Returns the rows where 0.7 > col > 0.5
* **df.sort\_values(col1) :**It sorts the values by col1 in ascending order.
* **df.sort\_values(col2,ascending=False) :**It sorts valus by col2 in descending order.
* **df.sort\_values([col1,col2],ascending=[True,False]) :**It sort the values by col1 in ascending order and col2 in descending order.
* **df.groupby(col1):** Returns a groupby object for the values from one column.
* **df.groupby([col1,col2]) :**Returns a groupby object for valus from multiple colmns.
* **df.groupby(col1)[col2]) :**Returns mean of the values in col2, grouped by the values in col1.
* **df.pivot\_table(index=col1,values=[col2,col3],aggfunc=mean) :**It creates the pivot table that groups by col1 and calculate mean of col2 and col3.
* **df.groupby(col1).agg(np.mean) :**It calculates the average across all the columns for every unique col1 group.
* **df.apply(np.mean) :**Its task is to apply function np.mean() across each column.
* **nf.apply(np.max,axis=1) :**Its task is to apply function np.max() across each row.

### Join/Combine

* **df1.append(df2):** Its task is to add the rows in df1 to the end of df2(columns should be identical).
* **pd.concat([df1, df2], axis=1):** Its task is to add the columns in df1 to the end of df2(rows should be identical).
* **df1.join(df2,on=col1,how='inner'):** SQL-style join the columns in df1 with the columns on df2 where the rows for col have identical values, 'how' can be of 'left', 'right', 'outer', 'inner'.

### Statistics: The statistics functions can be applied to a Series, which are as follows:

* **df.describe():** It returns the summary statistics for the numerical columns.
* **df.mean() :** It returns the mean of all the columns.
* **df.corr():** It returns the correlation between the columns in the dataframe.
* **df.count():** It returns count of all the non-null values in each dataframe column.
* **df.max():** It returns the highest value from each of the columns.
* **df.min():** It returns the lowest value from each of the columns.
* **df.median():** It returns the median from each of the columns.
* **df.std():** It returns the standard deviation from each of the columns.

# 

# Pandas Index

Pandas Index is defined as a vital tool that selects particular rows and columns of data from a DataFrame. Its task is to organize the data and to provide fast accessing of data. It can also be called a **Subset Selection**.

values are in **bold** font in the index, and the individual value of the index is called a **label**.

If we want to compare the data accessing time with and without indexing, we can use **%%timeit** for comparing the time required for various access-operations.

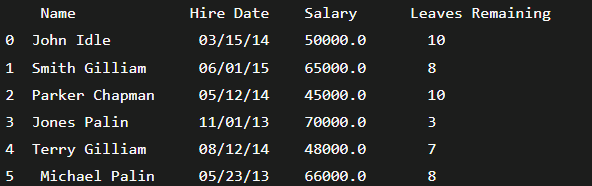
We can also define an index like an address through which any data can be accessed across the Series or DataFrame. A DataFrame is a combination of three different components, the **index**, **columns,** and the **data**.

### Axis and axes

An axis is defined as a common terminology that refers to rows and columns, whereas axes are collection of these rows and columns.

### Creating index:1st we have to take csv file that consist some data used for indxing.

1. data = pd.read\_csv("aa.csv")

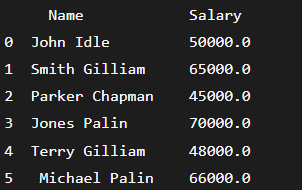


### Example1

1. info = pd.read\_csv("aa.csv", index\_col ="Name")
2. a = info[["Hire Date", "Salary"]] # retrieving multiple columns by indexing operator
3. print(a)



1. info =pd.read\_csv("aa.csv", index\_col ="Name")
2. a =info["Salary"] # retrieving columns by indexing operator
3. print(a)



### Set index

The '**set\_index**' is used to set the DataFrame index using existing columns. An index can replace the existing index and can also expand the existing index.

It set a list, Series or DataFrame as the index of the DataFrame.

1. info = pd.DataFrame({'Name': ['Parker', 'Terry', 'Smith', 'William'],
2. 'Year': [2011, 2009, 2014, 2010], 'Leaves': [10, 15, 9, 4]})
3. info
4. info.set\_index('Name')
5. info.set\_index(['year', 'Name'])
6. info.set\_index([pd.Index([1, 2, 3, 4]), 'year'])
7. a = pd.Series([1, 2, 3, 4])
8. info.set\_index([a, a\*\*2])



### Multiple Index: We can also have multiple indexes in the data.

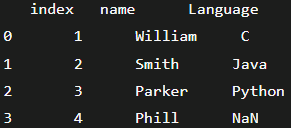
1. pd.MultiIndex(levels=[[np.nan, None, pd.NaT, 128, 2]], codes=[[0, -1, 1, 2, 3, 4]])



### Reset index:

### We can also reset the index using the '**reset\_index**' command. Let's look at the '**cm**' DataFrame again.

1. info = pd.DataFrame([('William', 'C'), ('Smith', 'Java'), ('Parker', 'Python'),
2. ('Phill', np.nan)], index=[1, 2, 3, 4], columns=('name', 'Language'))
3. info
4. info.reset\_index()

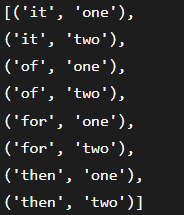


# Multiple Index

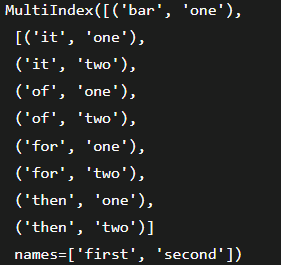
Multiple indexing is defined as a very essential indexing because it deals with the data analysis and manipulation, especially for working with higher dimensional data. It also enables to store and manipulate data with the arbitrary number of dimensions in lower dimensional data structures like Series and DataFrame.

It is the hierarchical analogue of the standard index object which is used to store the axis labels in pandas objects. It can also be defined as an array of tuples where each tuple is unique. It can be created from a list of arrays, an array of tuples, and a crossed set of iterables.

1. arrays = [['it', 'it', 'of', 'of', 'for', 'for', 'then', 'then'],
2. ['one', 'two', 'one', 'two', 'one', 'two', 'one', 'two']]
3. tuples = list(zip(\*arrays))
4. tuples



1. arrays = [['it', 'it', 'of', 'of', 'for', 'for', 'then', 'then'],
2. ['one', 'two', 'one', 'two', 'one', 'two', 'one', 'two']]
3. tuples = list(zip(\*arrays))
4. index = pd.MultiIndex.from\_tuples(tuples, names=['first', 'second'])



1. pd.MultiIndex(levels=[[np.nan, None, pd.NaT, 128, 2]],
2. codes=[[0, -1, 1, 2, 3, 4]])



# Reindex

The main task of the Pandas reindex is to conform DataFrame to a new index with optional filling logic and to place NA/NaN in that location where the values are not present in the previous index. It returns a new object unless the new index is produced as an equivalent to the current one, and the value of **copy** becomes **False**.

Reindexing is used to change the index of the rows and columns of the DataFrame. We can reindex the single or multiple rows by using the reindex() method. Default values in the new index are assigned NaN if it is not present in the DataFrame.

### Syntax:DataFrame.reindex(labels=None, index=None, columns=None, axis=None, method=None, copy=True, level=None, fill\_value=nan, limit=None, tolerance=None)

**labels:** It is an optional parameter that refers to the new labels or the index to conform to the axis that is specified by the 'axis'.

**index, columns :** It is also an optional parameter that refers to the new labels or the index. It generally prefers an index object for avoiding the duplicate data.

**axis :** It is also an optional parameter that targets the axis and can be either the axis name or the numbers.

**method:** It is also an optional parameter that is to be used for filling the holes in the reindexed DataFrame. It can only be applied to the DataFrame or Series with a monotonically increasing/decreasing order.

**None:** It is a default value that does not fill the gaps.

**pad / ffill:**It is used to propagate last valid observation forward to next valid observation.

**backfill / bfill:** To fill the gap, It uses the next valid observation.

**nearest:** To fill the gap, it uses the next valid observation.

**copy:** Its default value is True and returns a new object as a boolean value, even if the passed indexes are the same.

**level :** It is used to broadcast across the level, and match index values on the passed MultiIndex level.

**fill\_value :** Its default value is np.NaN and used to fill existing missing (NaN) values. It needs any new element for successful DataFrame alignment, with this value computation.

**limit :** It defines the maximum number of consecutive elements that are to be forward or backward fill.

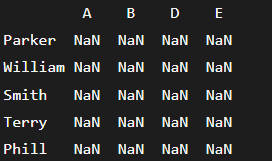
**tolerance :** It is also an optional parameter that determines maximum distance between original and new labels for inexact matches. At the matching locations, the values of the index should most satisfy the equation abs(index[indexer] ? target) <= tolerance.

### Returns : It returns reindexed DataFrame.

### Ex1:The below example shows the working of **reindex()** function to reindex the dataframe. In the new index,default values are assigned NaN in the new index that does not have corresponding records in the DataFrame.

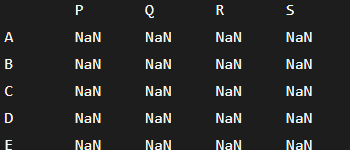
#### Note: We can use fill\_value for filling the missing values.

1. info = pd.DataFrame({"P":[4, 7, 1, 8, 9], "Q":[6, 8, 10, 15, 11], "R":[17, 13, 12, 16, 14],
2. "S":[15, 19, 7, 21, 9]}, index =["Parker", "William", "Smith", "Terry", "Phill"])
3. info



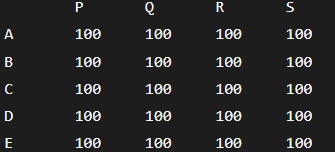
Now, we can use the dataframe.reindex() function to reindex the dataframe.

1. info.reindex(["A", "B", "C", "D", "E"]) # reindexing with **new** index values



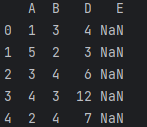
Notice that the new indexes are populated with NaN values. We can fill in the missing values using the fill\_value parameter.

1. info.reindex(["A", "B", "C", "D", "E"], fill\_value =100) #filling missing values by 100



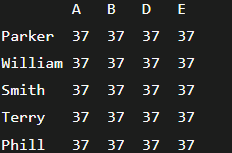
**Ex 2:** This example shows the working of reindex() function to reindex the column axis.

1. info1 =pd.DataFrame({"A":[1, 5, 3, 4, 2], "B":[3, 2, 4, 3, 4],
2. "C":[2, 2, 7, 3, 4], "D":[4, 3, 6, 12, 7]})
3. # reindexing the column axis with old and **new** index values
4. info.reindex(columns =["A", "B", "D", "E"])



Notice that NaN values are present in the new columns after reindexing, we can use the argument fill\_value to the function for removing the NaN values.

1. # reindex the columns fill the missing values by 37
2. info.reindex(columns =["A", "B", "D", "E"], fill\_value =37)



# Reset Index

The Reset index of the DataFrame is used to reset the index by using the '**reset\_index**' command. If DataFrame has a MultiIndex, this method can remove one or more levels.

**Syntax:** DataFrame.reset\_index(self, level=None, drop=False, inplace=False, col\_level=0, col\_fill='')

**level :** Refers to int, str, tuple, or list, default value None. It is used to remove the given levels from the index and also removes all levels by default.

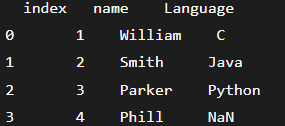
**drop :** Refers to Boolean value, default value False. It resets the index to the default integer index.

**inplace :** Refers to Boolean value, default value False. It is used to modify the DataFrame in place and does not require to create a new object.

**col\_level :** Refers to int or str, default value 0. It determines level the labels are inserted if the column have multiple labels

**col\_fill :** Refers to an object, default value ''. It determines how the other levels are named if the columns have multiple level.

1. info = pd.DataFrame([('William', 'C'), ('Smith', 'Java'), ('Parker', 'Python'),
2. ('Phill', np.nan)], index=[1, 2, 3, 4], columns=('name', 'Language'))
3. info
4. info.reset\_index()



# Set Index

Pandas set index() is used to set a List, Series or DataFrame as index of a Data Frame. We can set the index column while making a data frame. But sometimes a data frame is made from two or more data frames and then index can be changed using this method.

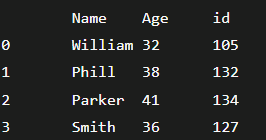
**Syntax:** DataFrame.set\_index(self, keys, drop=True, append=False, inplace=False, verify\_integrity=False)

* **keys:** Refers to label or array-like or list of labels/arrays. It can be either a single column key, a single array of the same length as the calling DataFrame, or also a list that contains an arbitrary combination of column keys and arrays.
* **drop:** Returns boolean value, default value is True. Used to delete the columns that are to be used as the new index.
* **append:** Returns the boolean value, default value is False. It checks whether append the columns to an existing index.
* **inplace:** Returns the boolean value, default value False. It is used to modify the DataFrame in place. We don't need to create a new object.
* **verify\_integrity:** Returns the boolean value, default value False. It checks the new index for duplicate values. Otherwise, it will defer the check until necessary. It also set it to False that will improve the performance of this method.

**Returns:** It will change the row labels as the output.

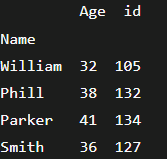
### Example1: This example shows how to set the index:

1. info = pd.DataFrame({'Name': ['William', 'Phill', 'Parker', 'Smith'],
2. 'Age': [32, 38, 41, 36], 'id': [105, 132, 134, 127]})
3. info



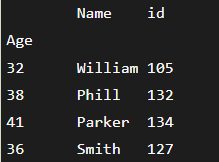
Now, we have to set the index to create the 'month' column:

1. info.set\_index('month')



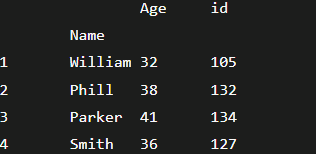
### Example2: Create a MultiIndex using columns 'Age' and 'Name':

1. info.set\_index(['Age', 'Name'])



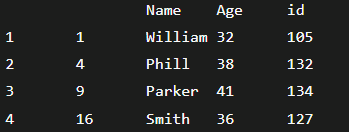
### Example3: It creates a MultiIndex using an Index and a column:

1. info.set\_index([pd.Index([1, 2, 3, 4]), 'Name'])



### Example4:Create a MultiIndex using two Series:

1. a = pd.Series([1, 2, 3, 4])
2. info.set\_index([a, a\*\*2])



# Pandas NumPy

Numerical Python (Numpy) is defined as a Python package used for performing the various numerical computations and processing of the multidimensional and single-dimensional array elements. The calculations using Numpy arrays are faster than the normal Python array.

# Pandas NumPy

Numerical Python (Numpy) is defined as a Python package used for performing the various numerical computations and processing of the multidimensional and single-dimensional array elements. The calculations using Numpy arrays are faster than the normal Python array.

It is also capable of handling a vast amount of data and convenient with Matrix multiplication and data reshaping.

Pandas are built over numpy array; therefore, numpy helps us to use pandas more effectively.

**Creating Arrays:** Main task of arrays is to store multiple values in a single variable. It defines multidimensional arrays that can be easily handled in numpy as shown in ex:-

1. **import** array
2. arr = array.array('l', [2, 4, 6, 8, 10, 12])# initializing array with array vals & signed int
3. print ("New created array: ",end="") # print the original array
4. **for** l in range (0,5):
5. print (arr[l], end=" ") //New created array: 2 4 6 8 10

### Boolean indexing

Boolean indexing is defined as a vital tool of numpy, which is frequently used in pandas. Its main task is to use the actual values of the data in the DataFrame. We can filter the data in the boolean indexing in different ways that are as follows:

* Access the DataFrame with a boolean index.
* Apply the boolean mask to the DataFrame.
* Masking the data based on column value.
* Masking the data based on the index value.

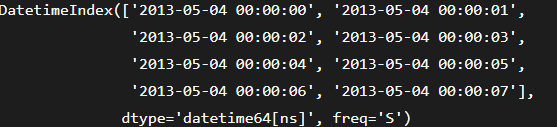
# Pandas Datetime

The Pandas can provide the features to work with time-series data for all domains. It also consolidates a large number of features from other Python libraries like scikits.timeseries by using the NumPy datetime64 and timedelta64 dtypes. It provides new functionalities for manipulating the time series data.

The time series tools are most useful for data science applications and deals with other

packages used in Python.

1. info = pd.date\_range('5/4/2013', periods = 8, freq ='S')



1. info = pd.DataFrame({'year': [2014, 2012], 'month': [5, 7], 'day': [20, 17]})
2. pd.to\_datetime(info) // 0 2014-05-20 && 1 2012-07-17

You can pass errors='ignore' if the date does not meet the timestamp. It will return the original input without raising any exception.

If you pass errors='coerce', it will force an out-of-bounds date to NaT.

1. **import** pandas as pd
2. pd.to\_datetime('18000706', format='%Y%m%d', errors='ignore')
3. datetime.datetime(1800, 7, 6, 0, 0)
4. pd.to\_datetime('18000706', format='%Y%m%d', errors='coerce')

//Timestamp('1800-07-06 00:00:00')

# Pandas Time Offset

The offset specifies a set of dates that conform to the DateOffset. We can create the DateOffsets to move the dates forward to valid dates.

If date is not valid, we can use rollback and rollforward methods for rolling the date to its nearest valid date before or after date. pseudo-code of time offsets are as follows:

### Syntax: **class** pandas.tseries.offsets.DateOffset(n=1, normalize=False, \*\*kwds)

**def \_\_add\_\_(date):**

date = rollback(date). It returns nothing if the date is valid + <n number of periods>.

**date = rollforward(date)**

When we create a date offset for a negative number of periods, the date will be rolling forward.

### Parameters: **n:** Refers to int, default value is 1. It is the number of time periods that represents the offsets.

**normalize:** Refers to a boolean value, default value False.

**\*\*kwds:** It is an optional parameter that adds or replaces the offset value.

The parameters used for **adding** to the offset are as follows:

* years, months, weeks, days, hours, minutes, seconds, microseconds, nanoseconds

The parameters used for **replacing** the offset value are as follows:

* year, month, day, weekday, hour, minute, second, microsecond, nanosecond

1. p = pd.Timestamp('2018-12-12 06:25:18') # Create the Timestamp
2. **do** = pd.tseries.offsets.DateOffset(n = 2) # Create the DateOffset
3. print(p) # Print the Timestamp
4. print(**do**) # Print the DateOffset



1. **import** pandas as pd
2. p = pd.Timestamp('2018-12-12 06:25:18') # Create the Timestamp
3. **do** = pd.tseries.offsets.DateOffset(n = 2) # Create the DateOffset
4. new\_timestamp = p + **do** # Add the dateoffset to given timestamp
5. print(new\_timestamp) //Timestamp('2018-12-14 06:25:18') #updated timestamp

# Pandas Time Periods

The Time Periods represent the time span, e.g., days, years, quarter or month, etc. It is defined as a class that allows us to convert the frequency to the periods.

### Generating periods and frequency conversion

We can generate the period by using '**Period**' command with frequency '**M**'. If we use '**asfreq**' operation with '**start**' operation, the date will print '**01**' whereas if we use the '**end**' option, the date will print '**31**'.

1. x = pd.Period('2014', freq='S')
2. x.asfreq('D', 'start')



1. x = pd.Period('2014', freq='S')
2. x.asfreq('D', 'end')



Period arithmetic

Period arithmetic is used to perform various arithmetic operation on periods. All the operations will be performed on the basis of '**freq**'.

1. x = pd.Period('2014', freq='Q')



1. x = pd.Period('2014', freq='Q')
2. x + 1



### Creating period range: We can create the range of period by using the '**period\_range**' command.



### Converting string-dates to period

If we want to Convert the string-dates to period, first we need to convert the string to date format and then we can convert the dates into the periods.

1. p = ['2012-06-05', '2011-07-09', '2012-04-06'] # dates as string
2. x = pd.to\_datetime(p) # convert string to date format



### Convert periods to timestamps

If we convert periods back to timestamps, we can simply do it by using '**to\_timestamp**' command.

# Convert string to date

The challenge behind this scenario is how the date strings are expressed. For example, 'Wednesday, June 6, 2018' can also be shown as '6/6/18' and '06-06-2018'. All these formats define the same date, but the code represents to convert each of them is slightly different.

1. from datetime **import** datetime
2. dmy\_str1 = 'Wednesday, July 14, 2018' # Define dates as the strings
3. dmy\_str2 = '14/7/17'
4. dmy\_str3 = '14-07-2017'
5. # Define dates as the datetime objects
6. dmy\_dt1 = datetime.strptime(date\_str1, '%A, %B %d, %Y')
7. dmy\_dt2 = datetime.strptime(date\_str2, '%m/%d/%y')
8. dmy\_dt3 = datetime.strptime(date\_str3, '%m-%d-%Y')
9. print(dmy\_dt1) //2017-07-14 00:00:00
10. print(dmy\_dt2) //2017-07-14 00:00:00
11. print(dmy\_dt3) //2018-07-14 00:00:00

### Converting the date string column

This conversion shows how to convert whole column of date strings from the dataset to datetime format.

From now on, you have to work with the DataFrame called **eth** that contains the historical data on ether, and also a cryptocurrency whose blockchain is produced by the Ethereum platform. The dataset consists the following columns:

* date: Defines the actual date, daily at 00:00 UTC.
* txVolume: It refers an unadjusted measure of the total value in US dollars, in outputs on the blockchain.
* txCount: It defines number of transactions performed on public blockchain.
* marketCap: Refers to the unit price in US dollars multiplied by the number of units in circulation.
* price: Refers an opening price in US dollars at 00:00 UTC.
* generatedCoins: Refers the number of new coins.
* exchangeVolume: Refers the actual volume which is measured by US dollars, at exchanges like GDAX and Bitfinex.